# 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, Raeed |
| 3 | Aaron, Jingming |
| 4 | Jingming, Raeed |
| 5 | Jingming, Raeed |

## 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 speed increases further |
| -60 | Backward speed increases further |
| -70 | Backward speed increases further |
| -80 | Backward speed increases further |
| -90 | Backward speed increases further |
| -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();
```

```c
    BOARD_InitBootClocks();

    setupPWM();

    /******* Delay *******/
    for (volatile int i = 0U; i < 1000000; i++)
        __asm("NOP");

    scanf("%d, %d", &motorInput, &servoInput);

    motorDutyCycle = motorInput * 0.025f / 100.0f + 0.070745;
    servoDutyCycle = servoInput * 0.025f / 45.0f + 0.078;

    updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, servoDutyCycle);
    updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, motorDutyCycle);

    FTM_SetSoftwareTrigger(FTM_MOTOR, true);

    while (1) {
    }
}
```

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.

```c
int main(void) {
  char buffer[] = {0, 0, 0, 0, 0, 0, 0, 0, 0};

  int motorInput;
  int servoInput;
  float motorDutyCycle;
  float servoDutyCycle;

  /* Init board hardware. */
  BOARD_InitBootPins();
  BOARD_InitBootClocks();
  BOARD_InitDebugConsole();

  setupUART();
  setupPWM();

  /******* Delay *******/
  for (volatile int i = 0U; i < 1000000; i++)
    __asm("NOP");

  char txbuff[] = "Hello World aaron\r\n";
  UART_WriteBlocking(TARGET_UART, txbuff, sizeof(txbuff) - 1);

  while (1) {
    UART_ReadBlocking(TARGET_UART, &buffer[0], 9);
```

```c

    sscanf(buffer, "%4d;%3d;", &motorInput, &servoInput);

    printf("%.9s\n%d %d\n", buffer, motorInput, servoInput);

    motorDutyCycle = motorInput * 0.025f / 100.0f + 0.070745;
    servoDutyCycle = servoInput * 0.025f / 45.0f + 0.078;

    updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, servoDutyCycle);
    updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, motorDutyCycle);

    FTM_SetSoftwareTrigger(FTM_MOTOR, true);
  }
}
```

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.

```c
void setupUART() {
    uart_config_t config;

    UART_GetDefaultConfig(&config);
    config.baudRate_Bps = 57600;
    config.enableTx = true;
    config.enableRx = true;
    config.enableRxRTS = true;
    config.enableTxCTS = true;
    UART_Init(TARGET_UART, &config, CLOCK_GetFreq(kCLOCK_BusClk));

    /******** Enable Interrupts *********/
    UART_EnableInterrupts(TARGET_UART, kUART_RxDataRegFullInterruptEnable);
    EnableIRQ(UART4_RX_TX_IRQn);
}
```

Listing 3: Problem 3 setupUART

```c
void UART4_RX_TX_IRQHandler() {
    UART_GetStatusFlags(TARGET_UART);
    ch = UART_ReadByte(TARGET_UART);

    PRINTF("GOT an interrupt from UART \n");

    if (new_char == 7) {
        return;
    }

    PRINTF("new_char = %d \n", new_char);
    PRINTF("ch = %c \n", ch);

    buffer[new_char] = ch;
    PRINTF(" inside the interrupt buffer[new_char] = %s \n ", buffer);
    new_char += 1;

}
```

Listing 4: Problem 3 UART4_RX_TX_IRQHandler

```c
int main(void) {
    char txbuff[] = "Hello World\r\n";

    int motorInput = 0;
    int servoInput = 0;
    float motorDutyCycle;
    float servoDutyCycle;
    /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    setupUART();
    setupPWM();

    /******* Delay *******/
    for (volatile int i = 0U; i < 1000000; i++)
        __asm("NOP");

    PRINTF("%s", txbuff);

    UART_WriteBlocking(TARGET_UART, txbuff, sizeof(txbuff) - 1);

    while (1) {
        if (new_char == 7) {
            new_char = 0;

            PRINTF("outside the interrupt buffer = %s \n ", buffer);

            PRINTF("cleared interrupt\n");

            sscanf(buffer, "%4d%3d", &motorInput, &servoInput);
            PRINTF("GOT input as motorInput = %d, servoInput = %d", motorInput, servoInput);
            motorDutyCycle = motorInput * 0.025f / 100.0f + 0.070745;
            servoDutyCycle = servoInput * 0.025f / 45.0f + 0.078;

            updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, servoDutyCycle);
            updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, motorDutyCycle);

            FTM_SetSoftwareTrigger(FTM_MOTOR, true);
            FTM_SetSoftwareTrigger(FTM_MOTOR, true);
        }
    }
}
```

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].

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

    masterTxData[0] = regAddress | 0x80; // Sets the most significant bit (enable write)
    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);
    free(masterRxData);

    return ret;
}
```

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.

```
int main(void)
{
    uint8_t byte;
    uint8_t write_test_byte = 0xCC;
    uint8_t read_test_byte;

    /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();

    voltageRegulatorEnable();
    accelerometerEnable();

    setupSPI();

    /******* Delay *******/
    for (volatile int i = 0U; i < 1000000; i++)
        __asm("NOP");

    SPI_read(0x0D, &byte, 1);
    printf("The expected value is 0xC7 and the read value 0x%X\n", byte);

    printf("reading  to the PL_COUNT register  before writing \n");
    SPI_read(0x12, &read_test_byte, 1);
    printf("The expected value for PL_COUNT register is 0x0 and the read value 0x%X\n",
        read_test_byte);

    SPI_write(0x12, write_test_byte);
    printf("writing to the PL_COUNT register \n");

    SPI_read(0x12, &read_test_byte, 1);
    printf("The expected value for the PL_COUNT register is 0xCC and the read value 0x%X\n",
         read_test_byte);

    while (1)
    {
```

```
    }
}
```

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 dspi 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.

```
int main(void)
{
    fxos_config_t config = {0};

    uint16_t i              = 0;
    uint16_t loopCounter    = 0;
    double sinAngle         = 0;
    double cosAngle         = 0;
    double Bx               = 0;
    double By               = 0;
```

```
    uint8_t array_addr_size = 0;

    status_t result;

    /* Board pin, clock, debug console init */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();

    voltageRegulatorEnable();
    accelerometerEnable();

    setupSPI();

    HW_Timer_init();

    /***** Delay *****/
    for (volatile uint32_t i = 0; i < 4000000; i++)
        __asm("NOP");


    /* Configure the SPI function */
    config.SPI_writeFunc = SPI_write;
    config.SPI_readFunc = SPI_read;

    result = FXOS_Init(&g_fxosHandle, &config);
    if (kStatus_Success != result)
    {
        PRINTF("\r\nSensor device initialize failed!\r\n");
    }

    /* Get sensor range */
    if (kStatus_Success != FXOS_ReadReg(&g_fxosHandle, XYZ_DATA_CFG_REG, &g_sensorRange, 1))
    {
        PRINTF("\r\nGet sensor range failed!\r\n");
    }

    switch (g_sensorRange)
    {
        case 0x00:
            g_dataScale = 2U;
            break;
        case 0x01:
            g_dataScale = 4U;
            break;
        case 0x10:
            g_dataScale = 8U;
            break;
        default:
            break;
    }

    PRINTF("\r\nTo calibrate Magnetometer, roll the board on all orientations to get max and min values\r\n");
    PRINTF("\r\nPress any key to start calibrating...\r\n");
    GETCHAR();
    Magnetometer_Calibrate();

    /* Infinite loops */
    for (;;)
    {
        if (SampleEventFlag == 1) /* Fix loop */
        {
            SampleEventFlag = 0;
            g_Ax_Raw        = 0;
            g_Ay_Raw        = 0;
            g_Az_Raw        = 0;
            g_Ax            = 0;
```

```
            g_Ay              = 0;
            g_Az              = 0;
            g_Mx_Raw          = 0;
            g_My_Raw          = 0;
            g_Mz_Raw          = 0;
            g_Mx              = 0;
            g_My              = 0;
            g_Mz              = 0;

            /* Read sensor data */
            Sensor_ReadData(&g_Ax_Raw, &g_Ay_Raw, &g_Az_Raw, &g_Mx_Raw, &g_My_Raw, &
                g_Mz_Raw);

            /* Average accel value */
            for (i = 1; i < MAX_ACCEL_AVG_COUNT; i++)
            {
                g_Ax_buff[i] = g_Ax_buff[i - 1];
                g_Ay_buff[i] = g_Ay_buff[i - 1];
                g_Az_buff[i] = g_Az_buff[i - 1];
            }

            g_Ax_buff[0] = g_Ax_Raw;
            g_Ay_buff[0] = g_Ay_Raw;
            g_Az_buff[0] = g_Az_Raw;

            for (i = 0; i < MAX_ACCEL_AVG_COUNT; i++)
            {
                g_Ax += (double)g_Ax_buff[i];
                g_Ay += (double)g_Ay_buff[i];
                g_Az += (double)g_Az_buff[i];
            }

            g_Ax /= MAX_ACCEL_AVG_COUNT;
            g_Ay /= MAX_ACCEL_AVG_COUNT;
            g_Az /= MAX_ACCEL_AVG_COUNT;

            if (g_FirstRun)
            {
                g_Mx_LP = g_Mx_Raw;
                g_My_LP = g_My_Raw;
                g_Mz_LP = g_Mz_Raw;
            }

            g_Mx_LP += ((double)g_Mx_Raw - g_Mx_LP) * 0.01;
            g_My_LP += ((double)g_My_Raw - g_My_LP) * 0.01;
            g_Mz_LP += ((double)g_Mz_Raw - g_Mz_LP) * 0.01;

            /* Calculate magnetometer values */
            g_Mx = g_Mx_LP - g_Mx_Offset;
            g_My = g_My_LP - g_My_Offset;
            g_Mz = g_Mz_LP - g_Mz_Offset;

            /* Calculate roll angle g_Roll (-180deg, 180deg) and sin, cos */
            g_Roll    = atan2(g_Ay, g_Az) * RadToDeg;
            sinAngle = sin(g_Roll * DegToRad);
            cosAngle = cos(g_Roll * DegToRad);

            /* De-rotate by roll angle g_Roll */
            By   = g_My * cosAngle - g_Mz * sinAngle;
            g_Mz = g_Mz * cosAngle + g_My * sinAngle;
            g_Az = g_Ay * sinAngle + g_Az * cosAngle;

            /* Calculate pitch angle g_Pitch (-90deg, 90deg) and sin, cos*/
            g_Pitch  = atan2(-g_Ax, g_Az) * RadToDeg;
            sinAngle = sin(g_Pitch * DegToRad);
            cosAngle = cos(g_Pitch * DegToRad);

            /* De-rotate by pitch angle g_Pitch */
```

```c
            Bx = g_Mx * cosAngle + g_Mz * sinAngle;

            /* Calculate yaw = ecompass angle psi (-180deg, 180deg) */
            g_Yaw = atan2(-By, Bx) * RadToDeg;
            if (g_FirstRun)
            {
                g_Yaw_LP    = g_Yaw;
                g_FirstRun = false;
            }

            g_Yaw_LP += (g_Yaw - g_Yaw_LP) * 0.01;

            if (++loopCounter > 10)
            {
                PRINTF("\r\nCompass Angle: %3.1lf", g_Yaw_LP);
                loopCounter = 0;
            }
        }
    } /* End infinite loops */
}
```

Listing 8: Problem 5 main