4DS4 Lab1 Report:
Bare-Metal I/O (Sensors and Actuators)

Johnnathan Gershkovich - 400408809
Joshua Umansky - 400234265
Shanzeb Immad - 400382928

Name	Contribution
Johnnathan Gershkovich	Wrote problems 5 writeup, Wrote problems 4&5 code
Joshua Umansky	Wrote problems 1,2,3 writeup, Co wrote problems 1, 2, 3 & 4 code
Shanzeb Immad	Wrote problem 4 writeup, Co wrote problems 1, 2 & 3 code

Experiment #1

1.14:

Duty Cycle	Observation
0	Fast and reverse
10	All 4 wheels are going forward at a slower speed
20	All 4 wheels forward and faster than duty cycle 10
30	All 4 wheels forward and faster than duty cycle 20
40	All 4 wheels forward and faster than duty cycle 30
50	All 4 wheels forward and faster than duty cycle 40
60	All 4 wheels forward and faster than duty cycle 50
70	All 4 wheels forward and faster than duty cycle 60
80	All 4 wheels forward and faster than duty cycle 70
90	All 4 wheels forward and faster than duty cycle 80
100	All 4 wheels forward and faster than duty cycle 90
-10	All 4 wheels going backwards at a decent speed
-20	All 4 wheels going backwards at an increasing speed relative

	to the test before
-30	All 4 wheels going backwards at an increasing speed relative to the test before
-40	All 4 wheels going backwards at an increasing speed relative to the test before
-50	All 4 wheels going backwards at an increasing speed relative to the test before
-60	All 4 wheels going backwards at an increasing speed relative to the test before
-70	All 4 wheels going backwards at an increasing speed relative to the test before
-80	All 4 wheels going backwards at an increasing speed relative to the test before
-90	All 4 wheels going backwards at an increasing speed relative to the test before
-100	All 4 wheels going backwards at an increasing speed relative to the test before

Problem #1:

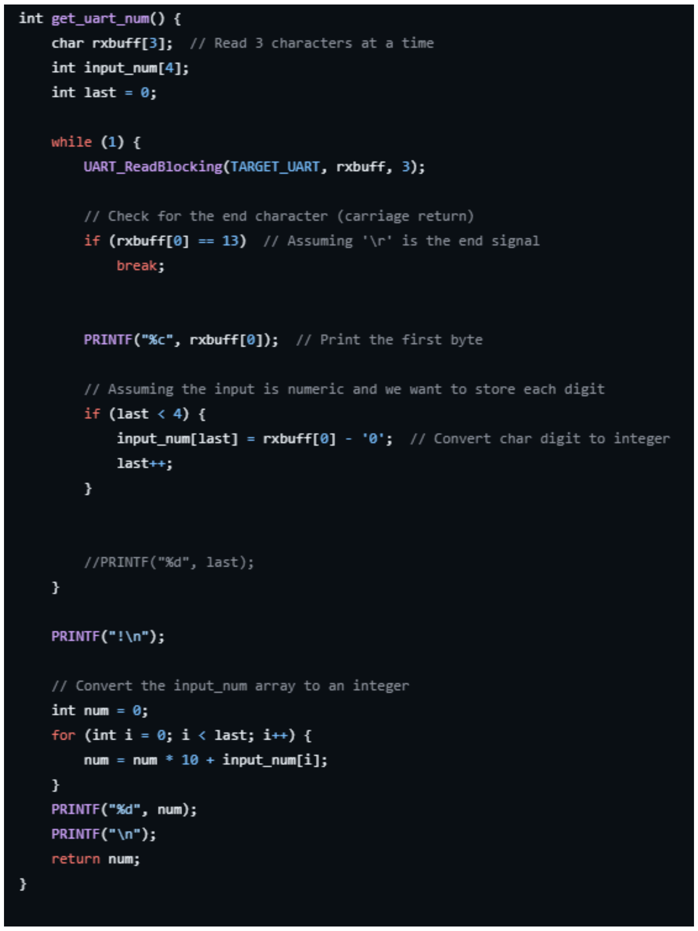
The code from experiment 1 was modified to include the second column of the J4 pin, this led our group to utilize Port A pin 6, and to use the Alt pin 3 to access FTMo_ch3. Another main change was adding a copy function of the setup pwm, one is the same as experiment 1 but changed for the servo and another for the motor. This allows the motor to spin along with being put at an angle. The duty cycle servo formula is defined as the following,

$$Duty\ Cycle\ = \frac{(\theta+90)*0.05}{180}\ +\ 0.05$$

With theta being any input angle from -90 to 90 degrees. -90 degrees represents turning full right, 0 being straight and 90 being full right.

Problem #2:

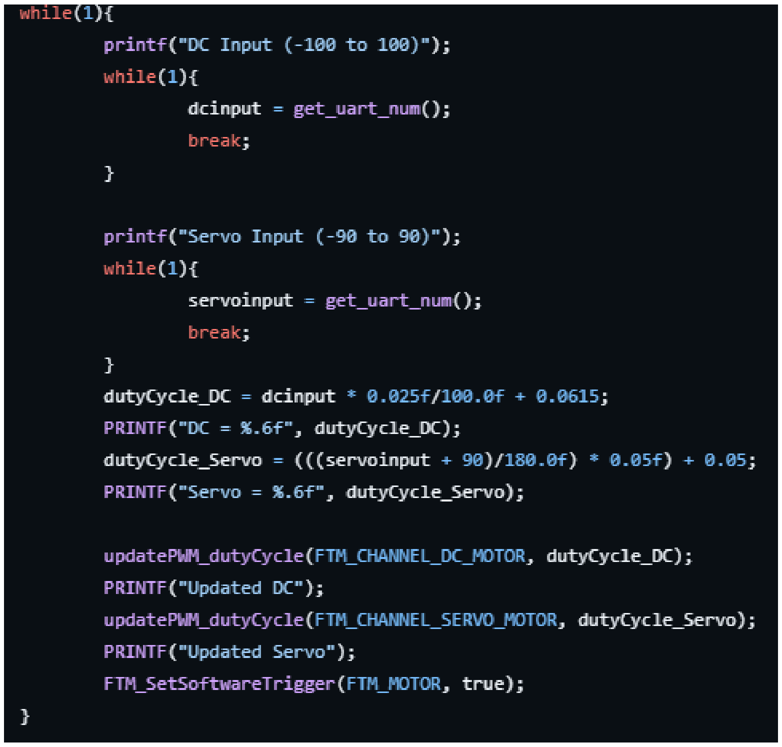
After implementing the supporting UART setup functions required from Experiment 2, we created a custom function that would read from the UART buffer for any input, up until the input was the enter key (Ascii 13). At that point the numbers entered into the buffer would be converted into an integer, and returned to the called line.



```c
int get_uart_num() {
    char rxbuff[3];  // Read 3 characters at a time
    int input_num[4];
    int last = 0;

    while (1) {
        UART_ReadBlocking(TARGET_UART, rxbuff, 3);

        // Check for the end character (carriage return)
        if (rxbuff[0] == 13)  // Assuming '\r' is the end signal
            break;


        PRINTF("%c", rxbuff[0]);  // Print the first byte

        // Assuming the input is numeric and we want to store each digit
        if (last < 4) {
            input_num[last] = rxbuff[0] - '0';  // Convert char digit to integer
            last++;
        }


        //PRINTF("%d", last);
    }

    PRINTF("!\n");

    // Convert the input_num array to an integer
    int num = 0;
    for (int i = 0; i < last; i++) {
        num = num * 10 + input_num[i];
    }
    PRINTF("%d", num);
    PRINTF("\n");
    return num;
}
```

Get_uart_num function

This function was then called twice, once for the DC input, and once for the Servo input, which are then converted similarly to Problem 1. These converted values are then sent to the PWM update cycles. In order to facilitate anytime updating, these function calls are placed into a while loop that runs endlessly.

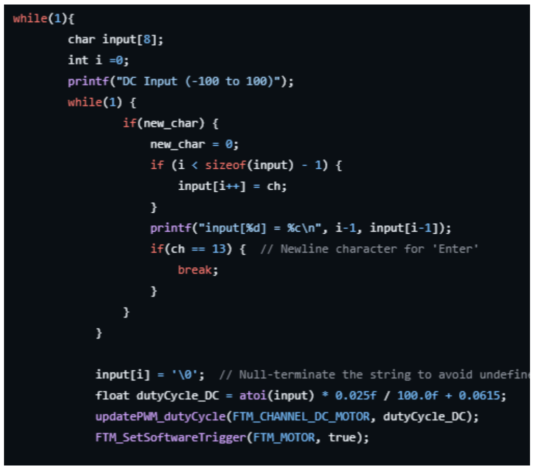


```c
while(1){
        printf("DC Input (-100 to 100)");
        while(1){
                dcinput = get_uart_num();
                break;
        }

        printf("Servo Input (-90 to 90)");
        while(1){
                servoinput = get_uart_num();
                break;
        }
        dutyCycle_DC = dcinput * 0.025f/100.0f + 0.0615;
        PRINTF("DC = %.6f", dutyCycle_DC);
        dutyCycle_Servo = (((servoinput + 90)/180.0f) * 0.05f) + 0.05;
        PRINTF("Servo = %.6f", dutyCycle_Servo);

        updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, dutyCycle_DC);
        PRINTF("Updated DC");
        updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, dutyCycle_Servo);
        PRINTF("Updated Servo");
        FTM_SetSoftwareTrigger(FTM_MOTOR, true);
}
```

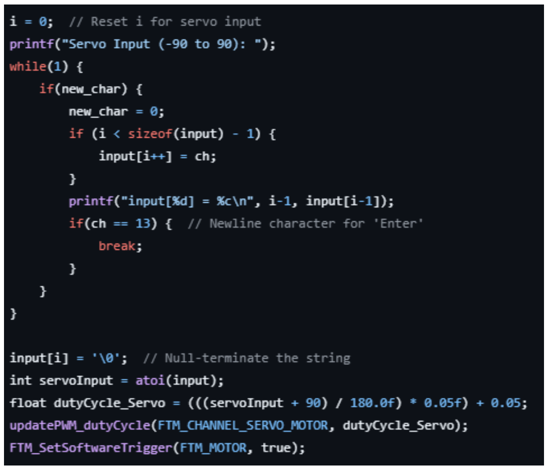
Main function of Problem 2

Problem #3:

After the interrupt handler was added from Experiment 3, and interrupts were enabled, we were able to implement UART interrupts to the previous code base. In order to do this, we created two while loops, then waiting for an interrupt to happen for each of the required inputs (DC and Servo).



```c
while(1){
        char input[8];
        int i =0;
        printf("DC Input (-100 to 100)");
        while(1) {
                if(new_char) {
                    new_char = 0;
                    if (i < sizeof(input) - 1) {
                        input[i++] = ch;
                    }
                    printf("input[%d] = %c\n", i-1, input[i-1]);
                    if(ch == 13) {  // Newline character for 'Enter'
                        break;
                    }
                }
        }

        input[i] = '\0';  // Null-terminate the string to avoid undefin
        float dutyCycle_DC = atoi(input) * 0.025f / 100.0f + 0.0615;
        updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, dutyCycle_DC);
        FTM_SetSoftwareTrigger(FTM_MOTOR, true);
```

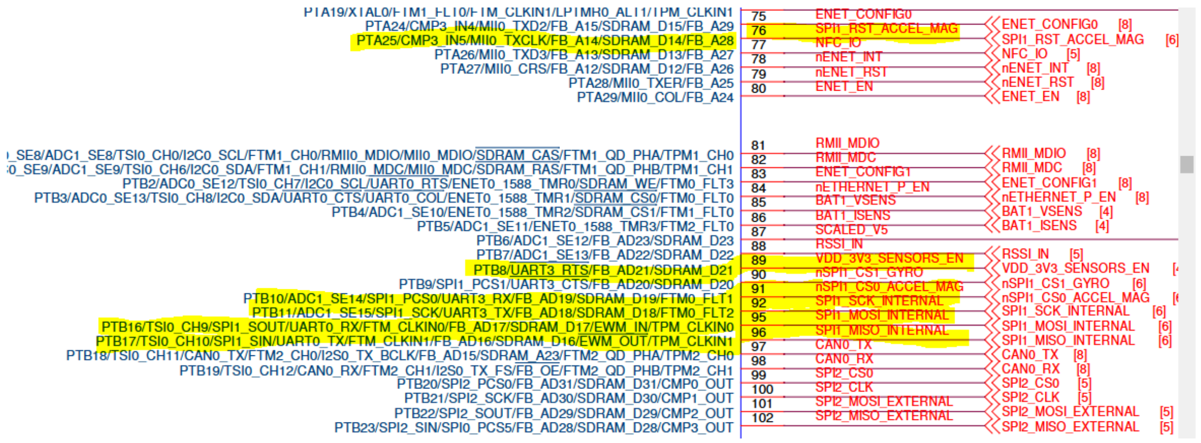


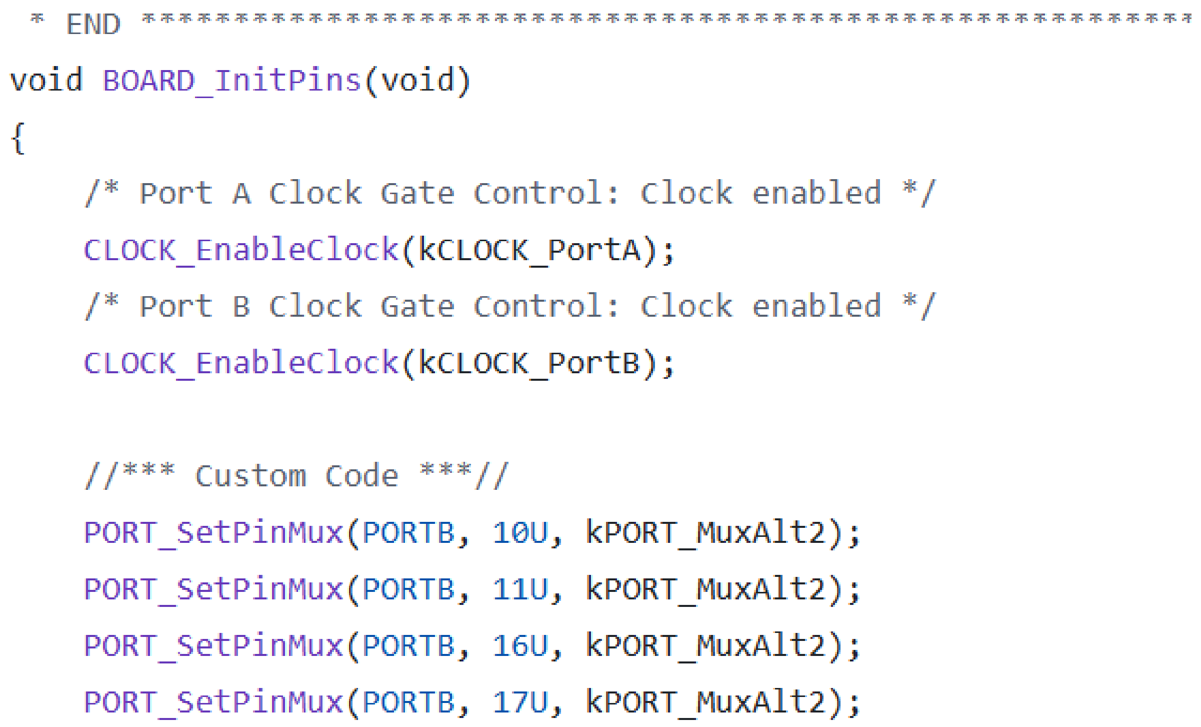
```c
i = 0;  // Reset i for servo input
printf("Servo Input (-90 to 90): ");
while(1) {
    if(new_char) {
        new_char = 0;
        if (i < sizeof(input) - 1) {
            input[i++] = ch;
        }
        printf("input[%d] = %c\n", i-1, input[i-1]);
        if(ch == 13) {  // Newline character for 'Enter'
            break;
        }
    }
}

input[i] = '\0';  // Null-terminate the string
int servoInput = atoi(input);
float dutyCycle_Servo = (((servoInput + 90) / 180.0f) * 0.05f) + 0.05;
updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, dutyCycle_Servo);
FTM_SetSoftwareTrigger(FTM_MOTOR, true);
```

Both loops wait for characters to be entered, and break the loop when the enter key is pressed. For debugging purposes, the value entered at each input is printed to the console to verify that the UART transmission was successful.

Problem #4:

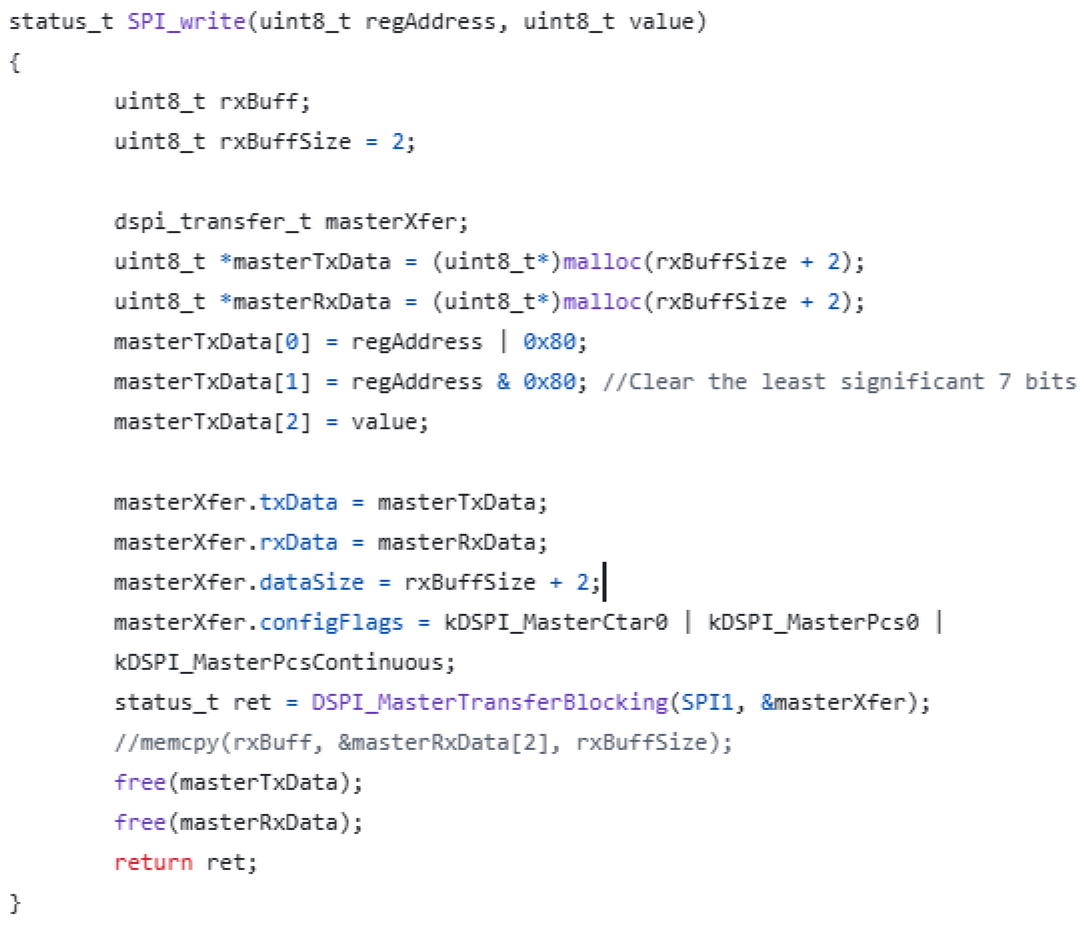
Objective of this problem was to implement SPI write to the experiment 4A completed before. The same pin arrangement was used as the experiment of 4A which shows the pins utilized below.


PTA19/XTAL0/FTM1_FLT0/FTM_CLKIN1/LPTMR0_ALT1/TPM_CLKIN1
PTA24/CMP3_IN4/MII0_TXD2/FB_A15/SDRAM_D15/FB_A29
PTA25/CMP3_IN5/MII0_TXCLK/FB_A14/SDRAM_D14/FB_A28
PTA26/MII0_TXD3/FB_A13/SDRAM_D13/FB_A27
PTA27/MII0_CRS/FB_A12/SDRAM_D12/FB_A26
PTA28/MII0_TXER/FB_A25
PTA29/MII0_COL/FB_A24
_SE8/ADC1_SE8/TSI0_CH0/I2C0_SCL/FTM1_CH0/RMII0_MDIO/MII0_MDIO/SDRAM_CAS/FTM1_QD_PHA/TPM1_CH0
0_SE9/ADC1_SE9/TSI0_CH6/I2C0_SDA/FTM1_CH1/RMII0_MDC/MII0_MDC/SDRAM_RAS/FTM1_QD_PHB/TPM1_CH1
PTB2/ADC0_SE12/TSI0_CH7/I2C0_SCL/UART0_RTS/ENET0_1588_TMR0/SDRAM_WE/FTM0_FLT3
PTB3/ADC0_SE13/TSI0_CH8/I2C0_SDA/UART0_CTS/UART0_COL/ENET0_1588_TMR1/SDRAM_CS0/FTM0_FLT0
PTB4/ADC1_SE10/ENET0_1588_TMR2/SDRAM_CS1/FTM1_FLT0
PTB5/ADC1_SE11/ENET0_1588_TMR3/FTM2_FLT0
PTB6/ADC1_SE12/FB_AD23/SDRAM_D23
PTB7/ADC1_SE13/FB_AD22/SDRAM_D22
PTB8/UART3_RTS/FB_AD21/SDRAM_D21
PTB9/SPI1_PCS1/UART3_CTS/FB_AD20/SDRAM_D20
PTB10/ADC1_SE14/SPI1_PCS0/UART3_RX/FB_AD19/SDRAM_D19/FTM0_FLT1
PTB11/ADC1_SE15/SPI1_SCK/UART3_TX/FB_AD18/SDRAM_D18/FTM0_FLT2
PTB16/TSI0_CH9/SPI1_SOUT/UART0_RX/FTM_CLKIN0/FB_AD17/SDRAM_D17/EWM_IN/TPM_CLKIN0
PTB17/TSI0_CH10/SPI1_SIN/UART0_TX/FTM_CLKIN1/FB_AD16/SDRAM_D16/EWM_OUT/TPM_CLKIN1
PTB18/TSI0_CH11/CAN0_TX/FTM2_CH0/I2S0_TX_BCLK/FB_AD15/SDRAM_A23/FTM2_QD_PHA/TPM2_CH0
PTB19/TSI0_CH12/CAN0_RX/FTM2_CH1/I2S0_TX_FS/FB_OE/FTM2_QD_PHB/TPM2_CH1
PTB20/SPI2_PCS0/FB_AD31/SDRAM_D31/CMP0_OUT
PTB21/SPI2_SCK/FB_AD30/SDRAM_D30/CMP1_OUT
PTB22/SPI2_SOUT/FB_AD29/SDRAM_D29/CMP2_OUT
PTB23/SPI2_SIN/SPI0_PCS5/FB_AD28/SDRAM_D28/CMP3_OUT
75 ENET_CONFIG0
76 SPI1_RST_ACCEL_MAG
77 NFC_IO
78 nENET_INT
79 nENET_RST
80 ENET_EN
81 RMII_MDIO
82 RMII_MDC
83 ENET_CONFIG1
84 nETHERNET_P_EN
85 BAT1_VSENS
86 BAT1_ISENS
87 SCALED_V5
88 RSSI_IN
89 VDD_3V3_SENSORS_EN
90 nSPI1_CS1_GYRO
91 nSPI1_CS0_ACCEL_MAG
92 SPI1_SCK_INTERNAL
95 SPI1_MOSI_INTERNAL
96 SPI1_MISO_INTERNAL
97 CAN0_TX
98 CAN0_RX
99 SPI2_CS0
100 SPI2_CLK
101 SPI2_MOSI_EXTERNAL
102 SPI2_MISO_EXTERNAL
ENET_CONFIG0 [8]
SPI1_RST_ACCEL_MAG [6]
NFC_IO [5]
nENET_INT [8]
nENET_RST [8]
ENET_EN [8]
RMII_MDIO [8]
RMII_MDC [8]
ENET_CONFIG1 [8]
nETHERNET_P_EN [8]
BAT1_VSENS [4]
BAT1_ISENS [4]
RSSI_IN [5]
VDD_3V3_SENSORS_EN
nSPI1_CS1_GYRO [6]
nSPI1_CS0_ACCEL_MAG
SPI1_SCK_INTERNAL [6]
SPI1_MOSI_INTERNAL [6]
SPI1_MISO_INTERNAL [6]
CAN0_TX [8]
CAN0_RX [8]
SPI2_CS0 [5]
SPI2_CLK [5]
SPI2_MOSI_EXTERNAL [5]
SPI2_MISO_EXTERNAL [5]



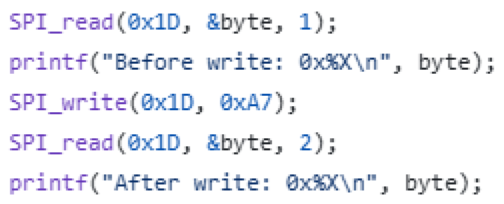
```
 * END ****************************************************
void BOARD_InitPins(void)
{
    /* Port A Clock Gate Control: Clock enabled */
    CLOCK_EnableClock(kCLOCK_PortA);
    /* Port B Clock Gate Control: Clock enabled */
    CLOCK_EnableClock(kCLOCK_PortB);

    //*** Custom Code ***//
    PORT_SetPinMux(PORTB, 10U, kPORT_MuxAlt2);
    PORT_SetPinMux(PORTB, 11U, kPORT_MuxAlt2);
    PORT_SetPinMux(PORTB, 16U, kPORT_MuxAlt2);
    PORT_SetPinMux(PORTB, 17U, kPORT_MuxAlt2);
```

To implement SPI write, the datasheet of the accelerometer was referred to, section 10.2.1 and 10.2.2. These sections revealed the byte format when write is called upon. The first byte has the most significant bit set to 1 and ⅞ bits of the address filling the rest. The second byte has the first bit set too the last address bit and the rest being ‘Don’t care’ values. Finally the final byte is filled with the value wished to be written. The function implements this by utilizing bitwise Or and AND operations, which only changes the masterTx as there is no read to receive(use rx) when we want to write.

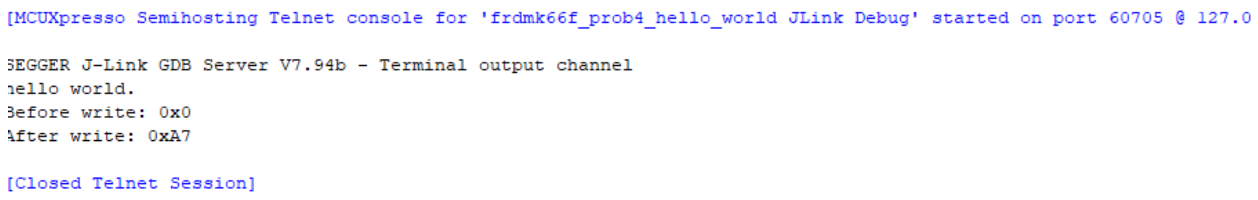


```
status_t SPI_write(uint8_t regAddress, uint8_t value)
{
        uint8_t rxBuff;
        uint8_t rxBuffSize = 2;

        dspi_transfer_t masterXfer;
        uint8_t *masterTxData = (uint8_t*)malloc(rxBuffSize + 2);
        uint8_t *masterRxData = (uint8_t*)malloc(rxBuffSize + 2);
        masterTxData[0] = regAddress | 0x80;
        masterTxData[1] = regAddress & 0x80; //Clear the least significant 7 bits
        masterTxData[2] = value;

        masterXfer.txData = masterTxData;
        masterXfer.rxData = masterRxData;
        masterXfer.dataSize = rxBuffSize + 2;
        masterXfer.configFlags = kDSPI_MasterCtar0 | kDSPI_MasterPcs0 |
        kDSPI_MasterPcsContinuous;
        status_t ret = DSPI_MasterTransferBlocking(SPI1, &masterXfer);
        //memcpy(rxBuff, &masterRxData[2], rxBuffSize);
        free(masterTxData);
        free(masterRxData);
        return ret;
}
```

The main function proceeds to be almost identical besides the SPI_write function call, chapter 14 of the datasheet aided in discovering which address to utilize to allow both read and write. Once run, the output shows what's read in register 1D before the write and after.



```
SPI_read(0x1D, &byte, 1);
printf("Before write: 0x%X\n", byte);
SPI_write(0x1D, 0xA7);
SPI_read(0x1D, &byte, 2);
printf("After write: 0x%X\n", byte);
```



```
[MCUXpresso Semihosting Telnet console for 'frdmk66f_prob4_hello_world JLink Debug' started on port 60705 @ 127.0

SEGGER J-Link GDB Server V7.94b - Terminal output channel
hello world.
Before write: 0x0
After write: 0xA7

[Closed Telnet Session]
```

Problem #5:

To start the setup of problem 5, the sdk demo example for an encompass peripheral device was imported and the dspi driver was added. Next, the following code was put in the main source file *ecompass_peripheral.c* from the code that was written in experiment #4 part A and B. This code is used to initialize the SPI communication and defines functions for reading and writing with SPI.

```c
void setupSPI()
{
    dspi_master_config_t masterConfig;
    /* Master config */
    masterConfig.whichCtar = kDSPI_Ctar0;
    masterConfig.ctarConfig.baudRate = 500000;
    masterConfig.ctarConfig.bitsPerFrame = 8U;
    masterConfig.ctarConfig.cpol = kDSPI_ClockPolarityActiveHigh;
    masterConfig.ctarConfig.cpha = kDSPI_ClockPhaseFirstEdge;
    masterConfig.ctarConfig.direction = kDSPI_MsbFirst;
    masterConfig.ctarConfig.pcsToSckDelayInNanoSec = 1000000000U / 500000;
    masterConfig.ctarConfig.lastSckToPcsDelayInNanoSec = 1000000000U / 500000;
    masterConfig.ctarConfig.betweenTransferDelayInNanoSec = 1000000000U / 500000;
    masterConfig.whichPcs = kDSPI_Pcs0;
    masterConfig.pcsActiveHighOrLow = kDSPI_PcsActiveLow;
    masterConfig.enableContinuousSCK = false;
    masterConfig.enableRxFifoOverWrite = false;
    masterConfig.enableModifiedTimingFormat = false;
    masterConfig.samplePoint = kDSPI_SckToSin0Clock;
    DSPI_MasterInit(SPI1, &masterConfig, BUS_CLK);
}

void voltageRegulatorEnable()
{
    gpio_pin_config_t pin_config = {
        .pinDirection = kGPIO_DigitalOutput,
        .outputLogic = 0U};
    GPIO_PinInit(GPIOB, 8, &pin_config);
    GPIO_PinWrite(GPIOB, 8, 1U);
}

void accelerometerEnable()
{
    gpio_pin_config_t pin_config = {
        .pinDirection = kGPIO_DigitalOutput,
        .outputLogic = 0U};
    GPIO_PinInit(GPIOA, 25, &pin_config);
    GPIO_PinWrite(GPIOA, 25, 0U);
}
```

```c
status_t SPI_read(uint8_t regAddress, uint8_t *rxBuff, uint8_t rxBuffSize)
{
    dspi_transfer_t masterXfer;
    uint8_t *masterTxData = (uint8_t*)malloc(rxBuffSize + 2);
    uint8_t *masterRxData = (uint8_t*)malloc(rxBuffSize + 2);
    masterTxData[0] = regAddress & 0x7F; //Clear the most significant bit
    masterTxData[1] = regAddress & 0x80; //Clear the least significant 7 bits
    masterXfer.txData = masterTxData;
    masterXfer.rxData = masterRxData;
    masterXfer.dataSize = rxBuffSize + 2;
    masterXfer.configFlags = kDSPI_MasterCtar0 | kDSPI_MasterPcs0 |
    kDSPI_MasterPcsContinuous;
    status_t ret = DSPI_MasterTransferBlocking(SPI1, &masterXfer);
    memcpy(rxBuff, &masterRxData[2], rxBuffSize);
    free(masterTxData);
    free(masterRxData);
    return ret;
}

status_t SPI_write(uint8_t regAddress, uint8_t value)
{
    uint8_t rxBuff;
    uint8_t rxBuffSize = 2;

    dspi_transfer_t masterXfer;
    uint8_t *masterTxData = (uint8_t*)malloc(rxBuffSize + 2);
    uint8_t *masterRxData = (uint8_t*)malloc(rxBuffSize + 2);
    masterTxData[0] = regAddress | 0x80;
    masterTxData[1] = regAddress & 0x80; //Clear the least significant 7 bits
    masterTxData[2] = value;

    masterXfer.txData = masterTxData;
    masterXfer.rxData = masterRxData;
    masterXfer.dataSize = rxBuffSize + 2;
    masterXfer.configFlags = kDSPI_MasterCtar0 | kDSPI_MasterPcs0 |
    kDSPI_MasterPcsContinuous;
    status_t ret = DSPI_MasterTransferBlocking(SPI1, &masterXfer);
    //memcpy(rxBuff, &masterRxData[2], rxBuffSize);
    free(masterTxData);
    free(masterRxData);
    return ret;
}
```

Later, in the main function, the read and write functions are added to the fxos_config_t struct which is then passed to the FXOS_init function.

```c
config.SPI_writeFunc = SPI_write;
config.SPI_readFunc = SPI_read;
.
.
```

```
.
result = FXOS_Init(&g_fxosHandle, &config);
```

In the *fsl_fxos.h* file the following two lines are added, to define spi read and write functions.

```
typedef status_t (*SPI_WriteFunc_t)(uint8_t regAddress, uint8_t value);
typedef status_t (*SPI_ReadFunc_t)(uint8_t regAddress, uint8_t *rxBuff, uint8_t
rxBuffSize);
```

Two structs in the same file, fxos_handle_t and fxos_config_t, were then modified to include two new members to store the read and write functions which were used above in config.SPI_writeFunc and so on.

```
/* Pointer to the user-defined SPI write Data function. */
SPI_WriteFunc_t SPI_writeFunc;
/* Pointer to the user-defined SPI read Data function. */
SPI_ReadFunc_t SPI_readFunc;
```

Next, in the *fsl_fxos.c* file, all instances of I2CsendFunc and I2CreadeFunc were replaced with the corresponding SPI functions. This allows all of the FXOS predefined code to be used but with our SPI read and write functions replacing their I2C ones.

Lastly, the same init pins and init clocks were defined.

```
CLOCK_EnableClock(kCLOCK_PortB);
/* Port C Clock Gate Control: Clock enabled */
CLOCK_EnableClock(kCLOCK_PortC);
/* Port E Clock Gate Control: Clock enabled */
CLOCK_EnableClock(kCLOCK_PortE);

CLOCK_EnableClock(kCLOCK_PortA);

//*** Custom Code ***//
PORT_SetPinMux(PORTB, 10U, kPORT_MuxAlt2);
PORT_SetPinMux(PORTB, 11U, kPORT_MuxAlt2);
PORT_SetPinMux(PORTB, 16U, kPORT_MuxAlt2);
PORT_SetPinMux(PORTB, 17U, kPORT_MuxAlt2);
```

```
// RESET PIN
PORT_SetPinMux(PORTA, 25U, kPORT_MuxAsGpio); //VDD
PORT_SetPinMux(PORTB, 8U, kPORT_MuxAsGpio); //RST
```

The rest of the code to interface with the magnetometer was prewritten in the example.