COMP ENG 4DS4

Lab 1 - Bare-Metal I/O (Sensors and Actuators)

Professor: Mohamed Hassan

Submitted on *February 3rd, 2025* by: GROUP 07

Aidan Mathew - mathea40 - 400306142

Aaron Rajan - rajana8 - 400321812

Sameer Shakeel - shakes4 - 400306710

Chris Jiang - jiangc46 - 400322193

Declaration of Contributions:

All students contributed to the lab report and did the write-up.

Student	Contributions
Aidan Mathew	Worked together on all problems, demo P1
Aaron Rajan	Worked together on all problems, demo P4
Chris Jiang	Worked together on all problems, demo P2
Sameer Shakeel	Worked together on all problems, demo P3

Experiments

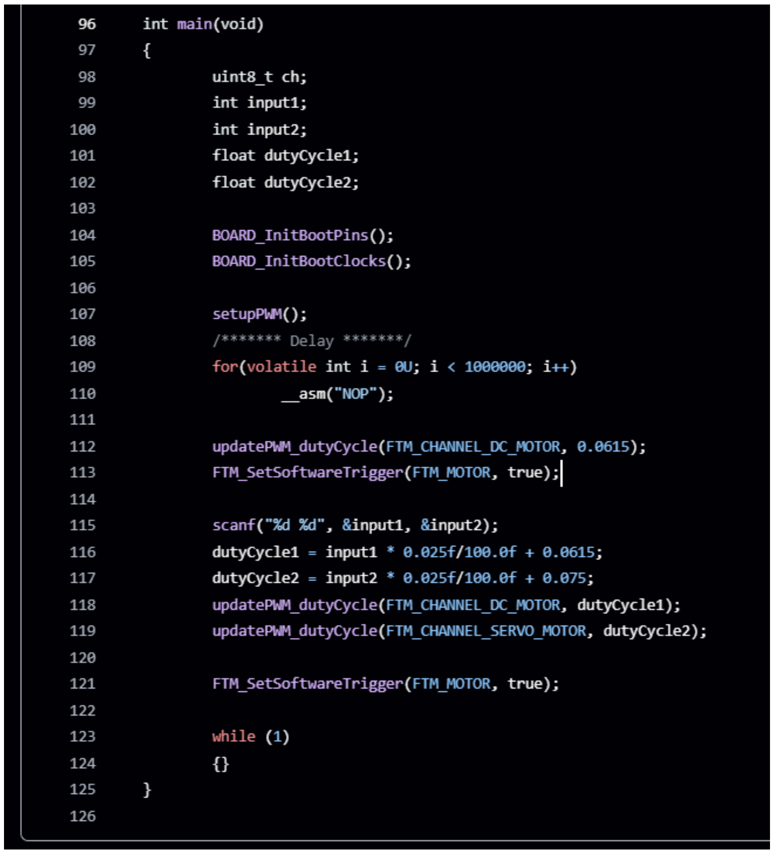
Experiment 1 - Control DC Motor

Try different duty cycles and record observations

Duty Cycle	Observations
0	N/A - no rotations
10	Clockwise - slow speed
20	Clockwise - faster than previous duty cycle
30	Clockwise - faster than previous duty cycle
40	Clockwise - faster than previous duty cycle
50	Clockwise - faster than previous duty cycle
60	Clockwise - faster than previous duty cycle
70	Clockwise - faster than previous duty cycle
80	Clockwise - faster than previous duty cycle
90	Clockwise - faster than previous duty cycle
100	Clockwise - faster than previous duty cycle
-10	CounterClockwise - slow speed
-20	CounterClockwise - faster than previous
-30	CounterClockwise - faster than previous
-40	CounterClockwise - faster than previous
-50	CounterClockwise - faster than previous
-60	CounterClockwise - faster than previous
-70	CounterClockwise - faster than previous
-80	CounterClockwise - faster than previous
-90	CounterClockwise - faster than previous
-100	Counter Clockwise - Fast

Problem 1

In problem 1, the primary objective was to control the speed of a DC motor and the steering angle of a servo motor using PWM signals generated by the FTM on the FMUK66 microcontroller. The code was designed to initialize and configure the FTM module to produce PWM signals with adjustable duty cycles, which directly influence the behaviour of the motors. For the DC motor, the duty cycle determines the speed, while for the servo motor, it controls the angular position of the front wheels. The initial duty cycles were set to 7% for both motors, providing a neutral starting point. User inputs were incorporated to allow dynamic adjustment of the duty cycles, with specific equations mapping the inputs to the standard PWM range of 5% to 10%. This range ensures precise control over the servo motor's angle and the DC motor's speed, leveraging the linear relationship between the duty cycle and motor response. Additionally, a delay loop was included to allow the PWM signals to stabilize after initialization, ensuring reliable operation. The use of software triggers ensures that updates to the duty cycles are applied immediately, enabling real-time control. Overall, the code demonstrates how PWM can be effectively used to manage multiple motors in a robotic or automotive system, providing a foundation for more complex control systems. See the below screenshot of the code from our GitHub repository:



```c
int main(void)
{
        uint8_t ch;
        int input1;
        int input2;
        float dutyCycle1;
        float dutyCycle2;

        BOARD_InitBootPins();
        BOARD_InitBootClocks();

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

        updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, 0.0615);
        FTM_SetSoftwareTrigger(FTM_MOTOR, true);

        scanf("%d %d", &input1, &input2);
        dutyCycle1 = input1 * 0.025f/100.0f + 0.0615;
        dutyCycle2 = input2 * 0.025f/100.0f + 0.075;
        updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, dutyCycle1);
        updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, dutyCycle2);

        FTM_SetSoftwareTrigger(FTM_MOTOR, true);

        while (1)
        {}
}
```

Problem 2
In problem 2, the task was to control a DC motor's speed and a servo motor's steering angle using PWM signals, similar to problem 1, but with inputs received via UART instead of the MCUXpresso console. The setupUART() function was added to configure the UART module for communication, setting a baud rate of 57600 and enabling both transmission and reception, along with hardware flow control (RTS and CTS) for reliable communication. The main() function was updated to handle UART-based input, starting with a welcome message, "Hello World," sent via UART to indicate system readiness. The program then prompts the user to enter values for the DC motor speed and servo motor angle, reading inputs character by character using UART_ReadBlocking() until the termination character 'x' is received. Each character is echoed back to the terminal for user feedback, and the input is stored in a buffer and converted to an integer using atoi(). These inputs are mapped to duty cycles using the equations dutyCycle1 = input1 * 0.025f / 100.0f + 0.0615 for the DC motor and dutyCycle2 = input2 * 0.025f / 100.0f + 0.075 for the servo motor, ensuring they fall within the standard PWM range of 5% to 10% for precise control. The PWM configuration and update logic remain unchanged, with the FTM module initialized to generate PWM signals and the updatePWM_dutyCycle() function adjusting the duty cycles based on the UART inputs. The use of UART simulates real-world communication, providing flexibility and enabling debugging through character echoing, making the program more versatile and aligned with practical embedded systems applications. This approach ensures reliable data transmission and lays the groundwork for more advanced communication protocols in future projects.

Problem 3

In Problem 3, the code from Problem 2 was updated to use UART interrupts for handling incoming data instead of polling, improving the program's efficiency and responsiveness. The setupUART() function was modified to enable UART interrupts by configuring the UART module to generate an interrupt when new data is received (kUART_RxDataRegFullInterruptEnable) and enabling the UART interrupt request (EnableIRQ(UART4_RX_TX_IRQn)). The UART4_RX_TX_IRQHandler() function was implemented to handle interrupts, reading the received byte and setting a flag (new_char) to indicate new data. In the main() function, the program waits for the new_char flag to be set, echoing the received character back to the terminal and storing it in an input buffer until a newline character ('\n', ASCII value 13) is received. This interrupt-driven approach allows the microcontroller to perform other tasks while waiting for UART input, reducing CPU usage and improving system responsiveness. The PWM configuration and update logic remained unchanged, with the FTM module generating PWM signals for the DC and servo motors, and the updatePWM_dutyCycle() function adjusting the duty cycles based on UART inputs. This modification demonstrates the benefits of using interrupts in embedded systems, making the program more efficient, scalable, and suitable for real-world applications.

Problem 4

In Problem 4, the task was to extend the previous SPI communication code to interact with a sensor's registers, specifically reading from and writing to them. This involved initializing the SPI interface, enabling the voltage regulator and accelerometer, and implementing functions to read from and write to the sensor's registers. The SPI_read() function reads data from a specified register address, while the SPI_write() function writes data to a specified register address. The program first reads the WHO_AM_I register (address 0x0D), which should return a constant value (0xC7), to verify the sensor's identity. It then writes a value (0x66) to register 0x13, reads it back to confirm the write, and repeats the process with a different value (0x88). This demonstrates the ability to configure and verify sensor settings through SPI communication. The use of blocking transfers ensures reliable data exchange, while the initialization of GPIO pins for the voltage regulator and accelerometer ensures proper power and communication setup. This implementation provides a foundation for more complex sensor interactions in embedded systems.

Problem 5

This code is designed to communicate with an accelerometer and magnetometer using the SPI (Serial Peripheral Interface) protocol, retrieve sensor data, and process it for further use. The setupSPI() function initializes the SPI module, setting it to Mode 3 with a 1 MHz clock speed to ensure reliable data transfer. Communication with the sensor is handled by SPI_write() and SPI_read(), where SPI_write() sends configuration commands to the sensor registers, and SPI_read() retrieves multiple bytes of data. The Sensor_ReadData() function is responsible for reading acceleration data by fetching six consecutive bytes from the sensor's registers, which contain the high and low bytes of the X, Y, and Z acceleration values. These bytes are combined using bitwise operations to form signed 16-bit integers representing raw acceleration readings. To improve accuracy, Magnetometer_Calibrate() performs sensor calibration by collecting minimum and maximum values for each axis over multiple samples and calculating offsets to remove bias. The program also utilizes a hardware timer, initialized with HW_Timer_init(), to generate an interrupt every millisecond, setting the SampleEventFlag variable, which signals when new sensor data should be acquired. The main() function begins by initializing the board and SPI interface, then puts the sensor into standby mode for configuration before setting it to active mode. The program then enters an infinite loop, where it continuously checks SampleEventFlag; when triggered, it reads and prints the latest acceleration values. This structured approach ensures efficient SPI-based sensor communication, periodic data acquisition, and accurate sensor readings for real-time motion analysis.