COMP ENG 4DS4

Lab 0 - Introduction Lab

Professor: Mohamed Hassan

Submitted on *January 21st, 2025* by: GROUP 07

Aidan Mathew - mathea40 - 400306142

Aaron Rajan - rajana8 - 400321812

Sameer Shakeel - shakes4 - 400306710

Chris Jiang - jiangc46 - 400322193

As a future member of the engineering profession, the student is responsible for performing the required work in an honest manner, without plagiarism and cheating. Submitting this work with my name and student number is a statement and understanding that this work is my own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario. **Submitted by Aidan Mathew, Aaron Rajan, Sameer Shakeel, and Chris Jiang.**

Declaration of Contributions:

Student	Contributions
Aidan Mathew - 400306142	Worked on Report, participated in all lab experiments, and worked on Problems 1, 2, 3, and 5.
Aaron Rajan -	Participated in all lab experiments and worked on Problems 1, 3, 4, 5.
Chris Jiang -	Helped with lab experiments and edited lab report.
Sameer Shakeel -	Participated in all lab experiments and worked on Problems 1, 3, 4, 5.

Lab Goals

- Get familiar with the tools (IDE, Debugger, board connection)
- Revise frequently used concepts in C
- Deal with memory-mapped registers
- Deal with the MCU's datasheet and the board's schematics
- Learn how to use GPIOs, UART and PWM

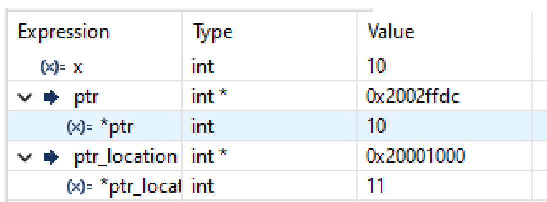
Experiments

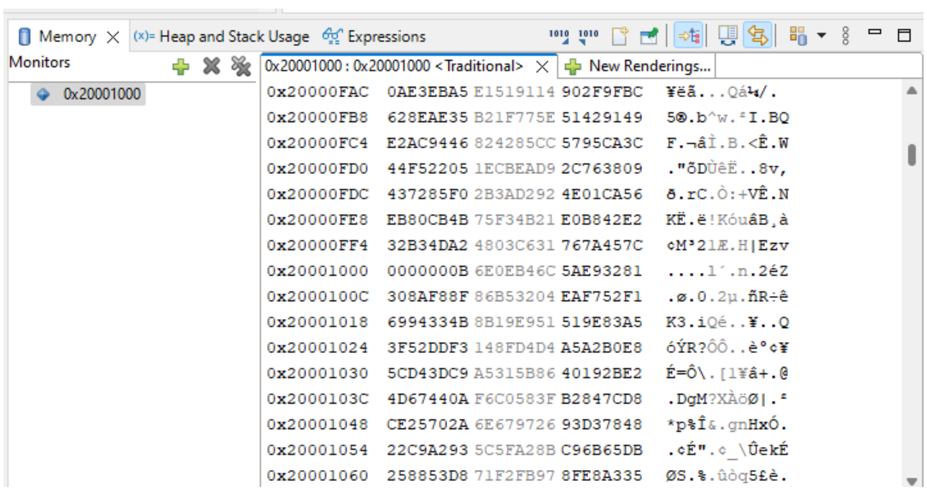
Experiment 1 - Hello World

In this first experiment we learned how to setup our environment and ran the hello world example.

Experiment 2 - C Pointers and Structures

During Part A of this experiment, we explored the use of pointers to access and manipulate memory on the microcontroller. A test function was implemented to demonstrate pointer behavior. An integer variable x was declared and initialized to 0, and a pointer ptr was assigned the address of x. Additionally, a second pointer, ptr_location, was assigned a direct memory address (0x20001000). The values at these locations were modified by dereferencing the pointers: *ptr was set to 10, and *ptr_location was set to 11. Debugging tools were used to verify the values. The expressions panel showed x with a value of 10, confirming that the pointer ptr correctly referenced x. Similarly, the value at memory address 0x20001000 was updated to 11 (the value B in Hex = 11), as shown in the memory panel screenshot. This experiment demonstrated the ability to access and modify specific memory locations on the MCU and the importance of debugging tools like the watch and memory panels for visualizing memory state and verifying pointer operations. See the below screenshots of the watch panel and the memory.

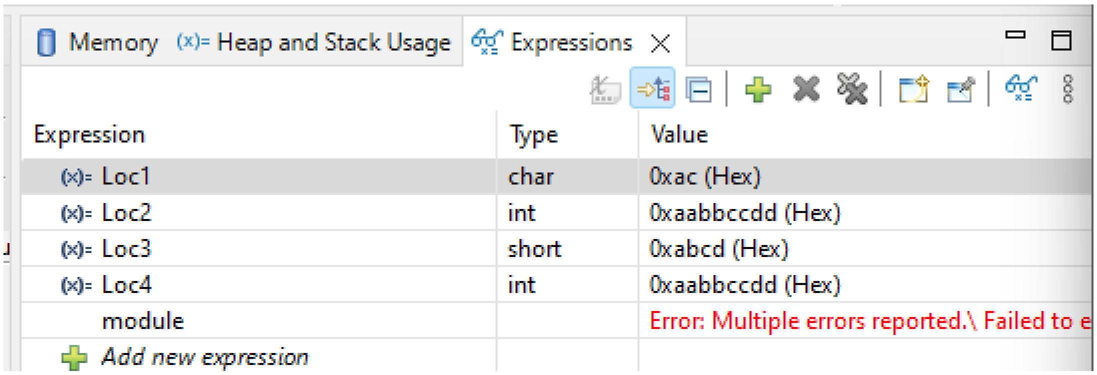


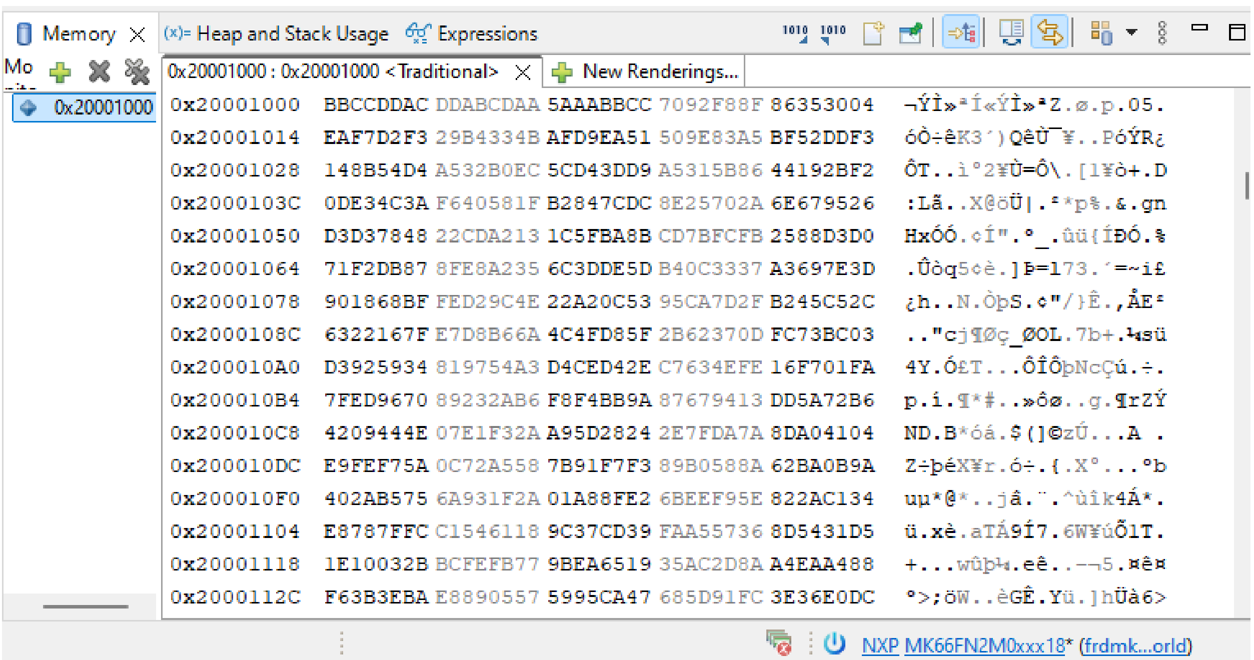


During part B of this experiment, we demonstrated how to efficiently access and manipulate specific memory locations using direct typecasting and dereferencing. Initially, a pointer (ptr_location) was declared and set to the memory address 0x20001000, where the value

11 was written. Using direct typecasting, the value 12 was written to another memory location, 0x20001004, without creating a separate pointer. The value at this location was then read and stored in the variable x. The screenshots validate the correctness of our implementation: the Expressions Panel shows x = 12, confirming that the value at 0x20001004 was successfully read, and ptr_location = 0x20001000, confirming the pointer's correct assignment. Additionally, the Memory Panel displays 11 at address 0x20001000 and 12 at 0x20001004, verifying that both memory locations were modified as intended. These results align with the experiment's objectives, demonstrating the efficiency of typecasting and the ability to manipulate memory on the MCU directly.

Problem 1:





In Problem 1, we addressed writing specific values to designated memory locations in the microcontroller's memory. To achieve this, we used macros to define memory locations (MEM_LOC, MEM_LOC_short, and MEM_LOC_char) with data types corresponding to the size of the required values. Each memory location (Loc1, Loc2, Loc3, and Loc4) was assigned its respective value as specified in the table. By leveraging macros, the code is concise and improves readability while ensuring proper handling of data types for memory alignment. The results, validated through memory inspection tools in the IDE, confirmed that the values were written correctly to the specified addresses with the appropriate configuration of Endianness, Cell Size, and Radix. This solution demonstrates efficient memory manipulation in embedded systems programming.

Problem 2:

The memory configuration used for analyzing the structures is Little Endian with a 4-byte Cell Size and values displayed in Hexadecimal Radix.

For struct1, the char x2 occupies 1 byte at offset 0. Since the following member, int x1, requires 4-byte alignment, the compiler adds 3 bytes of padding between x2 and x1, ensuring that x1 starts at offset 4. The total size of struct1 is 8 bytes, with 3 bytes of padding.

For struct2, the short x2 occupies 2 bytes at offset 0. The int x1 requires alignment to a 4-byte boundary, so the compiler adds 2 bytes of padding after x2, allowing x1 to start at offset 4. The total size of struct2 is also 8 bytes, with 2 bytes of padding.

For struct3, the int x1 occupies 4 bytes at offset 0, while the short x2 takes 2 bytes at offset 4. Since structures must be aligned to the size of their largest member (4 bytes in this case), the compiler adds 2 bytes of padding at the end of the structure, resulting in a total size of 8 bytes with 2 bytes of padding.

Finally, for struct4, the nested inner_struct comprises three members: char x1 (1 byte), short x2 (2 bytes with 1 byte of padding for alignment), and int x3 (4 bytes), making its size 8 bytes. The outer struct4 includes the inner_struct (8 bytes) and an int x1 (4 bytes) at offset 8, with no additional padding required. The total size of struct4 is 12 bytes, with no extra padding. See the table below for a clear definition of the structure, padding, and padding locations.

Structure	**Size (bytes)**	**Padding (bytes)**	**Padding Locations**
struct1	8	3	Between x2 (char) and x1 (int)
struct2	8	2	Between x2 (short) and x1 (int)
struct3	8	2	At the end of the structure
struct4	12	0	No padding needed

Problem 3:

In Problem 3, we modified the code to toggle three LEDs—LEDRGB_BLUE, LEDRGB_GREEN, and LEDRGB_RED—sequentially in the order BLUE → GREEN → RED, repeating in a loop. The changes involved configuring the necessary GPIO pins and toggling them using a delay loop to achieve a visible blinking effect.

In the BOARD_InitPins function, we enabled the clock for both Port C and Port D using the CLOCK_EnableClock function. We then configured the pins PTC8, PTC9, and PTD1 to operate as GPIO using the PORT_SetPinMux function. These pins correspond to the blue, green, and red LEDs, respectively.

In the main function, we initialized the LEDs as digital output pins using the GPIO_PinInit function. A simple delay function (delay()) was implemented using a for-loop with NOP instructions to introduce a delay between each LED toggle. The LEDs were toggled sequentially using the GPIO_PortToggle function, ensuring that each LED blinked on and off before moving to the next LED in the sequence. This process was repeated indefinitely in the main loop.

Problem 4:

In Problem 4, a custom GPIO driver was developed to replace the default SDK driver (fsl_gpio.h) for managing GPIO functionality. This driver consists of two key components: a header file (gpio_led_output.h) and a source file. The header file defines the GPIO_Types structure, which represents six key GPIO registers: PDOR, PSOR, PCOR, PTOR, PDIR, and PDDR. These registers are used to control GPIO operations such as setting, clearing, toggling, and reading pin values, as well as configuring pin directions. The header file also includes the prototypes for three helper functions: GPIOTogglePin, GPIOClearPin, and GPIOPinInit.

The GPIOTogglePin function toggles the logic state of a specific GPIO pin by writing to the PTOR register, while GPIOClearPin clears all pins in a given port by writing 0xFFFF to the PCOR register. The GPIOPinInit function configures a pin as input or output by modifying the PDDR register, using a configuration structure to specify the desired direction. These functions provide direct and efficient control over GPIO operations, bypassing the overhead of the higher-level SDK driver.

In the main function, the custom driver was used to toggle three LEDs—blue, green, and red—sequentially in the order BLUE (PTC8) → GREEN (PTC9) → RED (PTD1). The pins were first initialized as digital output using the GPIOPinInit function. A delay function was implemented to introduce a visible blinking effect, and the LEDs were toggled sequentially using GPIOTogglePin, with GPIOClearPin resetting the pins after each toggle. The process repeated in an infinite loop, demonstrating the effectiveness of the custom driver in managing GPIO operations.

This implementation highlights the importance of low-level programming in embedded systems by directly manipulating memory-mapped registers. The custom GPIO driver provides greater control over hardware, reduces abstraction layers, and enhances understanding of how embedded systems interact with peripherals. This approach not only optimizes performance but also serves as a practical learning experience for designing hardware-specific drivers.

Problem 5:

In Problem 5, we extended the use of Pulse-Width Modulation (PWM) to control the brightness of three LEDs—red, green, and blue—simultaneously, based on an RGB hex-code input from the user. The FlexTimer Module (FTM) was configured to generate PWM signals for each LED, allowing precise control of their brightness levels to produce a wide range of colors. The pins corresponding to the LEDs (PTD1, PTC8, and PTC9) were configured in the BOARD_InitPins function using the PORT_SetPinMux function to assign their alternative functionalities as PWM outputs. Each pin was mapped to a specific FTM channel: FTM3_CH1 for the red LED, FTM3_CH5 for the green LED, and FTM3_CH4 for the blue LED.

The pwm_setup function initialized the FTM for each LED by configuring the PWM parameters using the ftm_chnl_pwm_signal_param_t structure. The parameters included the channel number, a high-true PWM signal level, an initial duty cycle of 0%, and a frequency of 5 kHz, which ensured the LEDs did not flicker. The PWM signals were set up and started using the FTM_SetupPwm and FTM_StartTimer functions.

In the main function, the RGB hex-code input was read from the user, parsed into three separate integer values, and scaled from a range of 0 to 255 into percentages (0 to 100) to match the PWM duty cycle. The FTM_UpdatePwmDutycycle function was used to update the duty cycle of each LED, and a software trigger ensured the changes were applied immediately. For example, an input of FFFF00 resulted in a 100% duty cycle for the red and green LEDs and 0% for the blue LED, producing a yellow color. The program executed continuously in an infinite loop, maintaining the LED brightness levels based on the user-provided input.

This experiment demonstrated the practical application of PWM for controlling LED brightness and dynamically adjusting colors. It highlighted the use of the FTM module for generating independent PWM signals for multiple channels and showcased how hardware configurations, such as pin multiplexing and alternative functionalities, enable efficient control of peripheral devices in embedded systems. By incorporating an intuitive RGB hex-code interface, the implementation provided a user-friendly way to manage LED colours dynamically.