LAB 0
Introduction Lab

---

1 Lab Rules

- You have to stick to your Lab slot assigned to you on Mosaic.
- You have to form a team of 3 or 4 members at the beginning of the semester, and you cannot change your team afterward. **Those did not email me their group information: please do this ASAP**.
- Prepare a demonstration for all the Lab experiments, and get ready to be asked in any part of the experiments.
- The demonstrations of Lab 0 will be held starting from **Jan 23 at the first hour of each lab slot**.
- All the activities and questions written in **blue** should be documented and answered in your lab report.
- Each team needs to submit one report for all the members, and the first page of the report should contain the team number and the names of its members.
- The report should be submitted before **23:59 Jan 27**. Put the report in a PDF format, name it with your group number (will be assigned this week on Avenue once all of you form their groups), and submit it to Avenue through the dropbox that will open for the lab on avenue.
- The first page (After the title page) of the report must include a **Declaration of Contributions**, where each team member writes his own individual tasks conducted towards the completion of the lab.

General Note:

- Make sure to get all your work from the lab computer before leaving the lab room because your saved work may be deleted after the lab slot.

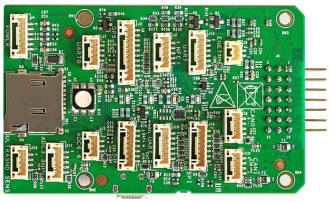
2 Lab Goals

- Get familiar with the tools (IDE, Debugger, board connection, etc.).
- Revise some of the 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.

3 Lab Components

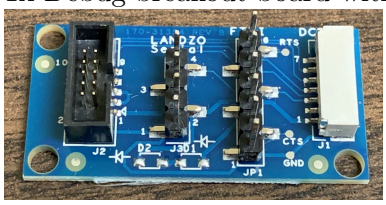
These are the components you will be working with during the lab. All of them have already been integrated into the embedded system; your job is to connect the system to the computer and program it, as we will explain below.

1. 1x RDDRONE-FMUK66 board.



2. 1x Segger J-Link EDU Mini debugger.

3. 1x Debug breakout board with the 7 wires cable.



4. 2x micro USB cables.

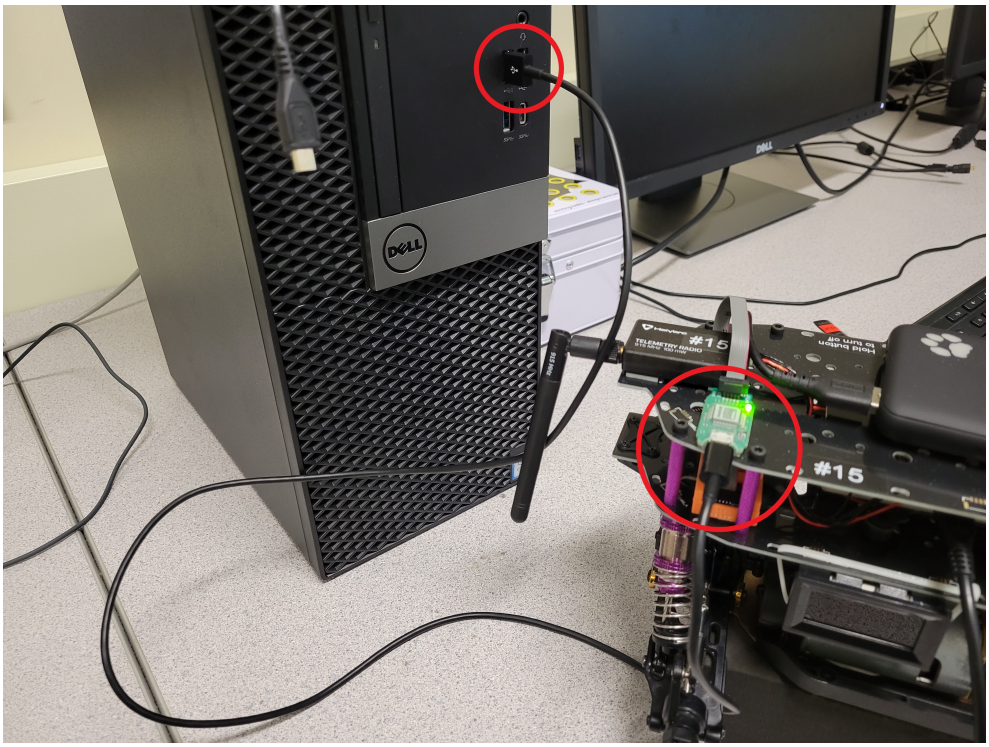
4 Experiments

Experiment 1: Hello World

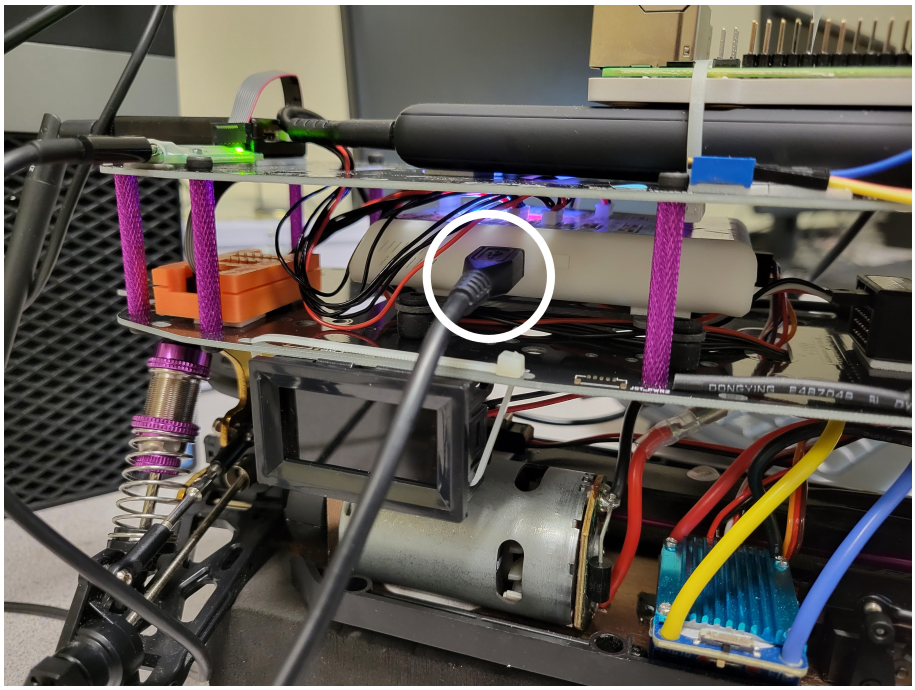
In this experiment, you are required to download a simple program to the FMUK66 board using the J-link debugger. The debugger's role in this experiment and all the following experiments is to download the program to the FMUK66 board, print messages coming from the board, and carry out the step-by-step debugging of the program. This program used in this experiment will send a "Hello World" message.

Experiment Setup

1. Connect the J-link debugger, which is located on the top of the device, to your lab PC via the micro-USB cable:



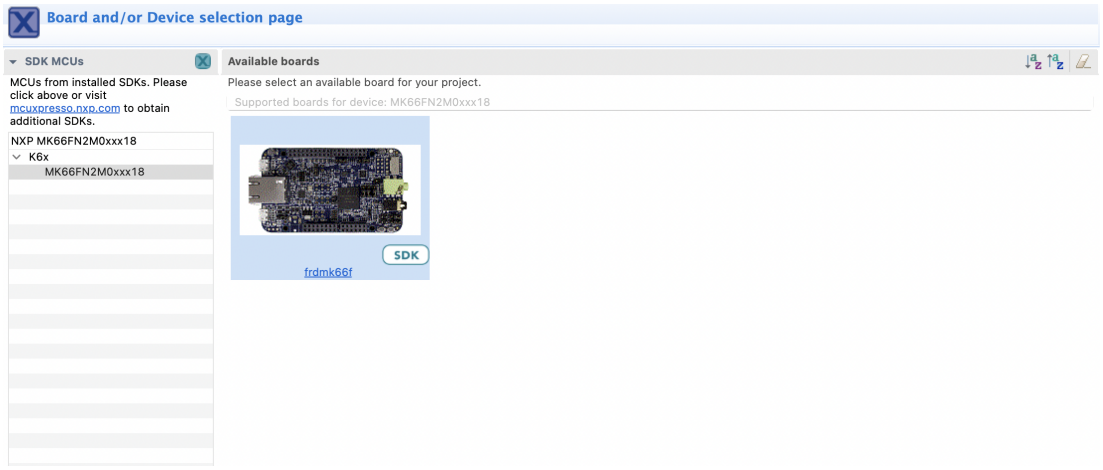
2. Connect the FMUK66 microcontroller board to the PC using the other micro-USB cable:



3. Open the MCUXpresso IDE.
   MCUXpresso is an IDE from NXP (the FMUK66 manufacturer); the IDE will give us the tools and code libraries to ease the interaction with FMUK66 and the debugger.

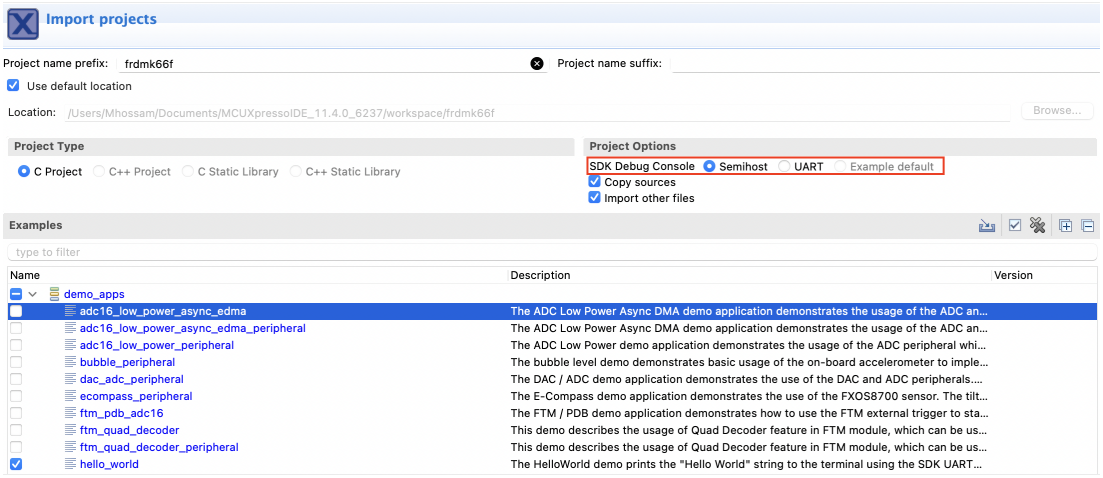
4. From the Quickstart Panel, click on "Import SDK example(s)..."

5. The IDE will open the Import Wizard from which choose frdmk66f SDK and click Next.



You may notice that the frdmk66f board is different from the board we use, FMUK66. However, both boards contain the same MCU, NXP MK66F, which makes the SDK 90% compatible with our board. Throughout the labs, we will learn how to deal with the discrepancies between the two boards and edit the SDK code accordingly.

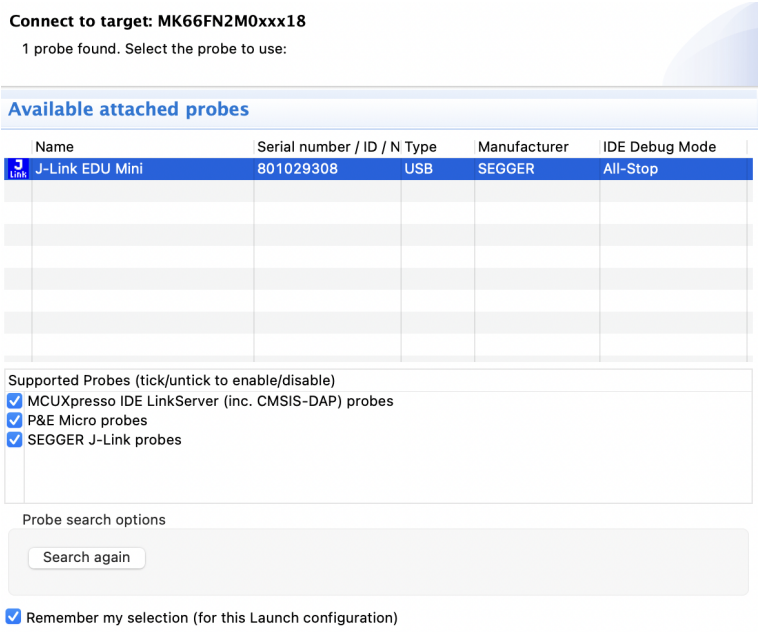
6. From the examples, choose demo_apps→hello_world. **Before clicking Finish, make sure to choose "Semihost" from Project Options**. This option will direct the print messages (any `printf` or `puts` functions) to the debugger instead of the UART. Always choose this option if you plan to print debugging messages from your code.



7. Select the created project from the Project Explorer, and then From Miscellaneous under the Quickstart Panel, select Quick Settings→SDK Debug Console→Semihost Console.

8. From the Quickstart Panel, click on "Debug" .

9. If your connections are correct, the IDE will discover the connected J-link and show that in "Probes discovered" window.

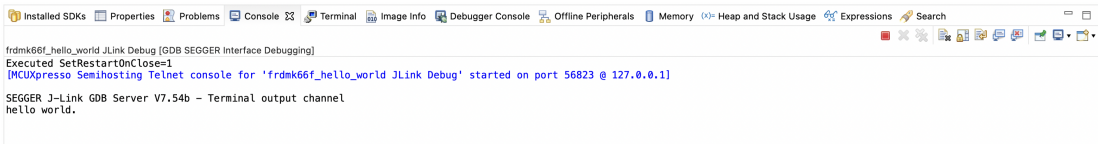


10. Press OK, and you will see the project starts to be built. You may also get a message from J-link to accept their terms of use. Once you accept the terms, you will see the code starts to be downloaded to the board; afterwards, the program execution will be paused in the beginning of the code.

```
30 int main(void)
31 {
32     char ch;
33
34     /* Init board hardware. */
35     BOARD_InitBootPins();
36     BOARD_InitBootClocks();
37     BOARD_InitDebugConsole();
38
39     PRINTF("hello world.\r\n");
40
41     while (1)
42     {
43         ch = GETCHAR();
44         PUTCHAR(ch);
45     }
46 }
```

11. From the Run menu, click on "Resume"  , or press F8.

12. In the Console, the print message appears, and the rest of the code waits for a character input from the Console to echo it.



13. To stop the program, click on "Terminate" from Run menu, or press CTRL + F2.

Experiment 2: C Pointers and Structures

This experiment will revise some essential C concepts that are commonly used in embedded software. By the end of this experiment, you should be able to read and write to any memory location and use debugging tools such as watch variables and the memory monitor.

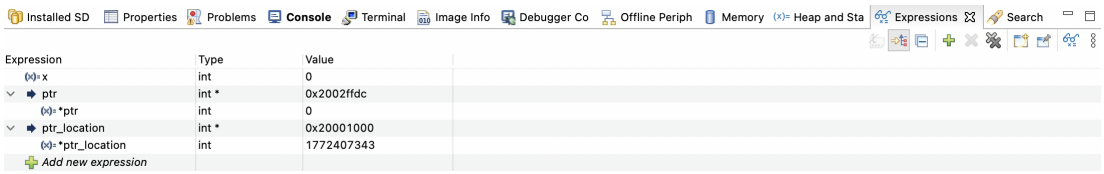
Experiment Setup: Part A

1. Use the same project of Experiment 1.
2. Write the following code in a separate function, and call this function before the while loop inside the main function.

```
1 void testFunction()
2 {
3     int x = 0;
4     int *ptr = &x;
5     int *ptr_location = 0x20001000; //Random location in the memory
6
7     *ptr = 10;
8     *ptr_location = 11;
9 }
```

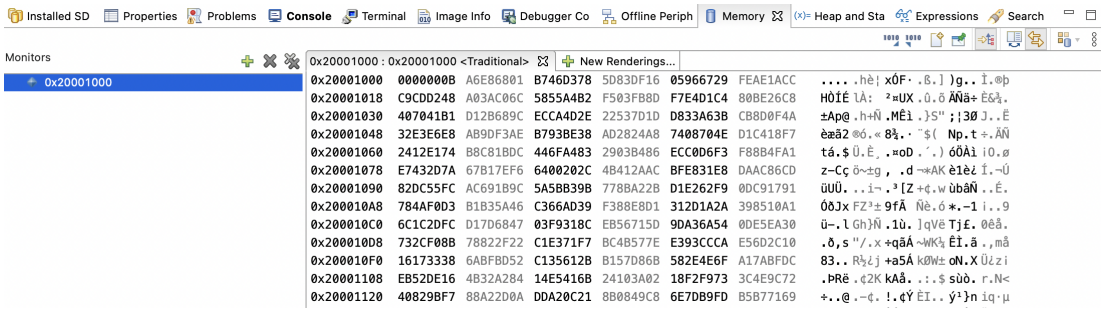
   This code creates an integer variable, `x`, in line 3, and in line 4, it creates an integer pointer that points at `x`. This is the traditional way of using pointers in C-code, and if you try to execute these two lines on your PC, they will run as expected. On the other hand, line 5 defines an integer pointer that points directly at a memory location defined by its address. Unlike the other two lines, this line will not run on your PC as easily as the traditional method, and that is because the operating systems do not allow user applications to access any memory locations as they wish. However in this case, the code will execute directly on the MCU without an operating system in the middle. Finally, the last two lines in the code verify that we can write to the two memory locations that `ptr` and `ptr_location` point at by dereferencing the pointers.

3. Call `testFunction` from the main function.
4. Build your code and make sure it is error-free.
5. Add a breakpoint at line 7 (`*ptr = 10`). In order to add breakpoints in MCUXpresso, double-click on the left of the line number and a blue breakpoint will appear on that line.
6. Let the code runs until it stops at the breakpoint.
7. Highlight `x`, right-click on it, and choose "Add Watch Expression..." to watch the value of the variable while executing.
8. Add both `ptr` and `ptr_location` to the expression panel.



9. Run two steps by clicking twice on the step over button , or navigate to "Run" menu and select "Step Over."

10. Navigate to the "Memory" panel and click on the add button to watch the memory starting from the input address. In the pop-up input box, write the memory that `ptr_location` points at.



The panel shows the data stored in the MCU's memory. Each row starts with an address specified in the beginning, and each column represents an offset from the row address. The columns offset is column number × cell size, while the column number starts from zero. By default, the cell size is 4 bytes, but it can be changed by right-clicking inside the memory panel and hovering over "Cell Size," then choosing the required size. Similarly, you can change the number of displayed columns from "Columns," the numbering system from "Radix," and the displayed endianness (the byte order of each cell) from "Endian."

11. Add a screenshot of the watch panel in your lab report that shows the value of the three variables, and a screenshot of the memory that shows the value of address 0x20001000.

Experiment Setup: Part B

1. Change `testFunction` to the following code.

```
1 void testFunction()
2 {
3     int *ptr_location = (int*)0x20001000;
4     *ptr_location = 11;
5
6     *((int*)0x20001004) = 12;
7     int x = *((int*)0x20001004);
8 }
```

Lines 3 and 4 are similar to lines 5 and 8 from Part A. They create a pointer to a memory location and then modify its value. This way of writing to a memory location is inefficient since we have to create a pointer then dereference it (it takes two steps).

It is possible to save this pointer and reduce writing (or reading) to memory to one step using typecasting, as shown in line 6. In order to understand line 6, break it down into two parts. Firstly, the part inside the parentheses that casts the numerical value (`0x20001004`) into an integer pointer. Secondly, the outside asterisk that dereferences the pointer created by the typecasting. Similarly, line 7 uses the same way to read the value of location `0x20001004` and store it in variable `x`.

2. Run the code and take screenshots from the resulting values in both expression and memory panels.

3. The code now is more efficient, yet hard to read. In order to improve the readability of your code, it is recommended to use `#define` to give names to the memory locations, as shown in the following code.

```
//It reads\writes the value of 0x20001004
#define ARBITRARY_LOCATION      *((int*)0x20001004)

void testFunction()
{
    int *ptr_location = (int*)0x20001000;
    *ptr_location = 11;

    ARBITRARY_LOCATION = 12;
    int x = ARBITRARY_LOCATION;
}
```

4. We can even improve the code more by creating a macro for the typecasting and the deference, as shown in the following code.

```
#define MEM_LOC(x)              *((int*)x)
#define ARBITRARY_LOCATION      MEM_LOC(0x20001004)

void testFunction()
{
    int *ptr_location = (int*)0x20001000;
    *ptr_location = 11;

    ARBITRARY_LOCATION = 12;
    int x = ARBITRARY_LOCATION;
}
```

C macros look like functions, but in fact their role is to inform the compiler to perform a text replacement before compiling the code. Therefore, the macro `MEM_LOC` is replaced by `*((int*)x)`, and `x` here is, also, replaced by the value between the parentheses (`0x20001004`.)

Problem 1

Write a code that writes the required values to the locations specified in the following table. You should use macros and `#define` in your code. Also, add a screenshot of the memory panel after executing that the code and mention the used configuration (Endianness, Cell Size, and

Radix). Note that the data sizes are different; your code should handle them differently.

Name	Address	Size	Required Value
Loc1	0x20001000	1 Byte	0xAC
Loc2	0x20001001	4 Bytes	0xAABBCCDD
Loc3	0x20001005	2 Bytes	0xABCD
Loc4	0x20001007	4 Bytes	0xAABBCCDD

Experiment Setup: Part C

We will continue working with pointers in this Part, but we will combine them with structures. Due to their modularity, structure pointers are frequently used in embedded systems to write software drivers for the MCU peripherals. One of the following experiments will introduce you to a simple driver implementation, and you may check the implementation of the drivers included in the frdmk66f SDK.

1. Create a new function, `testfunction2`, and write the following code.

```
struct ARBITRARY_MODULE
{
    int location_1;
    char location_2;
    int location_3;
};

void testFunction2()
{
    struct ARBITRARY_MODULE* module = (struct ARBITRARY_MODULE*)0x20001000;
    module->location_1 = 0xAAAAAAAA;
    module->location_2 = 0xBB;
    module->location_3 = 0xCCCCCCCC;
}
```

   This code assumes a module, `ARBITRARY_MODULE`, that comprises three contiguous memory locations. The first and last locations require 4-byte data; hence, `int` is the chosen data type to represent them. In contrast, `char` is used for the second location since it requires only one byte of data.

2. We can improve the code readability by using `typedef`, which it will save us from keep writing `struct` before the structure name. The syntax of `typedef` is as the following code.

```
typedef struct OPTIONAL_NAME
{
  //The elements of the structure
} TheMandatoryNewTypeName;
```

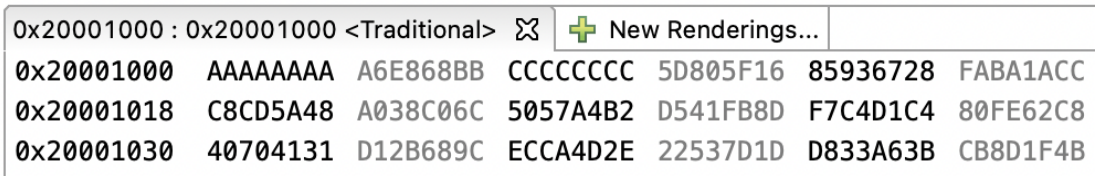
3. Another optimization is to `#define` instead of defining a pointer. Note, in case of using structure pointers, you do not need to add the dereferencing asterisk since the arrow, `->`, works as the dereferencing operator with structures. The final code is the following:

```
typedef struct
{
    int location_1;
    char location_2;
    int location_3;
}ARBITRARY_MODULE;

#define MODULE ((ARBITRARY_MODULE*)0x20001000)

void testFunction2()
{
    MODULE->location_1 = 0xAAAAAAAA;
    MODULE->location_2 = 0xBB;
    MODULE->location_3 = 0xCCCCCCCC;
}
```

4. Run the code after applying the optimizations, and check its correctness by using the memory panel.



It is clear from the values in the memory panel that there is an error in the execution of the code. The values should be similar to those in the table below, but the memory instead shows a 3-byte gap between `location_2` and `location_3`.

AAAAAAAA	CCCCCCBB	xxxxxxCC

The data alignment optimization is the reason for the discrepancy between the values that the memory shows and the expected values. The compiler tries to align the data in your program to 4-byte alignment, meaning the address of any 4-byte variable (or larger) should be divisible by 4. Therefore, in our structure, the compiler adds three extra bytes after `location_2`, so the address of `location_3` becomes divisible by 4. In order to verify this, try to print the size of the structure by using `sizeof(ARBITRARY_MODULE)`; it will print 12 instead of 9. The motive behind aligning the data is to improve the performance of the running code, and this should not affect the correctness of the execution. However, in our case, we are interested in mapping the locations correctly, not the performance. Consequently, we should disable this optimization on our structure. There is a preprocessor directive that directs the compiler to skip applying data alignment on the specified structure, `__attribute__((__packed__))`. The structure definition should look like the following:

```
typedef struct __attribute__((__packed__))
{
    int location_1;
    char location_2;
    int location_3;
}ARBITRARY_MODULE;
```

5. Rerun the code after adding the preprocessor and make sure that you get the correct result.

Problem 2

Calculate the sizes of the given structures and take into your account any extra bytes for alignment. Specify the number and locations of the extra bytes, if they exist.

```
struct struct1
{
    char x2;
    int x1;
};

struct struct2
{
    short x2;
    int x1;
};

struct struct3
{
    int x1;
    short x2;
};

struct struct4
{
    struct inner_struct
    {
        char x1;
        short x2;
        int x3;
    } inner_struct_1;
    int x1;
};
```

Experiment 3: General-Purpose Input/Output (GPIO)

In this experiment, we will use frdmk66f SDK to drive the LEDs on the FMUk66 board. In order to do this, we will use the GPIO example code from the SDK and configure it to be suitable for FMUK66 board.

Experiment Setup

1. Import from the SDK examples "driver_examples>gpio>gpio_led_output", and do not forget to choose "Semihost" for debugging.

2. Open the FMUK66 schematics, which are found in SPF-39053.pdf on Avenue.

3. Search for "D10 LED AMB", which is an amber led on the board. You may notice from the connection of the LED that it is active low, which means that it turns on when the pin outputs zero.

4. Get the name of the pin connected to the LED. The pin is PTD13 which indicates that it belongs to PortD and the pin number is 13.

5. Back to the code, change the definitions, `BOARD_LED_GPIO_PIN` and `BOARD_LED_GPIO`, to `13` and `GPIOD`, respectively.

6. Navigate to the definition of `BOARD_InitBootPins` function; after that, navigate to the definition of `BOARD_InitPins`. This can be done by holding Ctrl and clicking on the function name.

7. Inside `BOARD_InitPins`, there are a few things you should do before using any pin for any purpose. Firstly, enable the clock of the Port that contains the required pin. Secondly, configure the pin to work as a GPIO, and this step is mandatory since each pin has multiple functions and only one function can be enabled at a time. The different functions of the pins appear in the schematics in the pins' names, and also you can find them listed in Section 11.3.1 in the MCU datasheet, K66P144M180SF5RMV2.pdf. You can also find this file on Avenue.

8. The code of `BOARD_InitPins` should look like the following code:

```
void BOARD_InitPins(void)
{
    /* Port D Clock Gate Control: Clock enabled */
    CLOCK_EnableClock(kCLOCK_PortD);

    /* PORTD13 is configured as PTD13 */
    PORT_SetPinMux(PORTD, 13U, kPORT_MuxAsGpio);
}
```

9. Build the code and download it on the board. You should see the board LED blinking.

Problem 3

Repeat experiment 3, but instead of blinking "D10 LED AMB", toggle the three LEDs, "LEDRGB_BLUE," "LEDRGB_GREEN," and "LEDRGB_RED," sequentially such that it goes by the order BLUE→GREEN→RED and then repeats. Note that in the schematics, wires with the same name are connected even if they appear disconnected, as shown in Figures 1 and 2, "UI_LED_RED" is connected to PTD1.

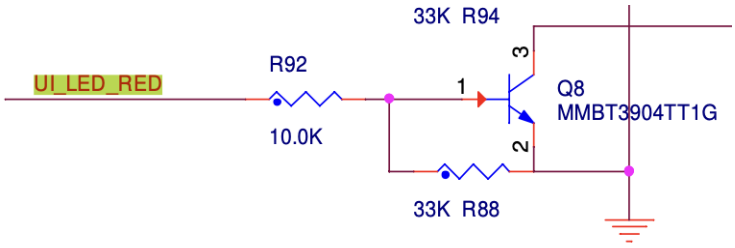


Figure 1: Example from FMUK66 board schematics.

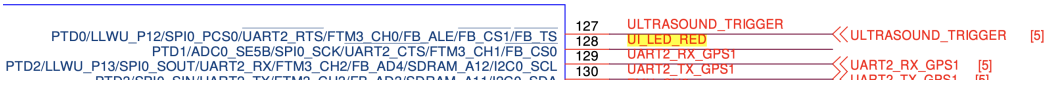


Figure 2: Example from FMUK66 board schematics.

Problem 4

Read Section 63.3 from the MCU datasheet, which explains the GPIO registers. You are required to write a simple driver for the GPIOs instead of the one in the SDK. This driver should contain a structure of the 6 GPIO registers, GPIOx_PDOR, GPIOx_PSOR, GPIOx_PCOR, GPIOx_PTOR, GPIOx_PDIR, and GPIOx_PDDR, as explained in Experiment 2: Part C. The driver should be written in two files, header file (.h) and source file (.c). The header file contains the definition of the structure and the prototypes of the driver's helper functions, while the source file contains the implementation of the helper functions. The helper functions are the function that facilitate the usage of the driver, for instance, you can provide a function that toggles a selected pin in a port as helper function, `togglePin(port, pin)`. Add at least 3 helper functions to your driver. Finally, rewrite the code of Problem 3 using this driver.

In order to organize the actions required by this problem, you may follow these steps:

1. Read Section 63.3 to understand the usage of each GPIO register, and figure out their memory locations' order.
2. Create a structure based on the sizes of the registers and their order. The structure should cover only the aforementioned registers.
3. Define the GPIO modules using the structure and their address. For instance, to define GPIOA, you should have a similar definition in the header file
   `#define GPIOA ((GPIO_Struct*) 0xXXXXXXXX)`
4. Add the helper functions. Note each helper function should take the target GPIO module as an input, and the type of this input is `GPIO_Struct*`.

Experiment 4: Pulse-Width Modulation (PWM)

The purpose of this experiment is to control the PWM of the FMUK66 board using the FRDMK66f SDK. Afterwards, we will use PWM to control the brightness of the UI_LED.

Experiment Setup

1. Import a new Hello World project, but you have to give it a new name if you have another Hello World project in your workspace.

2. Choose one of the three UI_LED and configure its pin inside the `BOARD_InitBootPins` function.

3. Unlike configuring GPIO pins, you can't configure the pin multiplexer (`PORT_SetPinMux` function) as "`kPORT_MuxAsGpio`", but you should choose the appropriate functionality. The module that is responsible for output the PWM signals is called FlexTimer Module (FTM), which is documented in Chapter 45 in the datasheet. Consequently, to configure the pin multiplexer with the correct value to assign the pin to FTM, you should check the table in section 11.3.1 which lists all the pins and their alternative functions. For instance, you may choose UI_LED_RED which is connected to PTD1, from the table, PTD1's fourth alternative function is FTM3_CH1 (Channel 1 of the third FlexTimer Module.) Accordingly, the pin mux of PTD1 should be assigned to "`kPORT_MuxAlt4`" while calling the `PORT_SetPinMux` function.

4. Back to hello_world.c, we need to change the code of `BOARD_InitBootClocks` as well, since PWM requires accurate timing, the clock configuration has to match the FMUK66 board. To change the code, navigate inside `BOARD_InitBootClocks` function, then inside `BOARD_BootClockRUN` function, and change the function code to the following code.

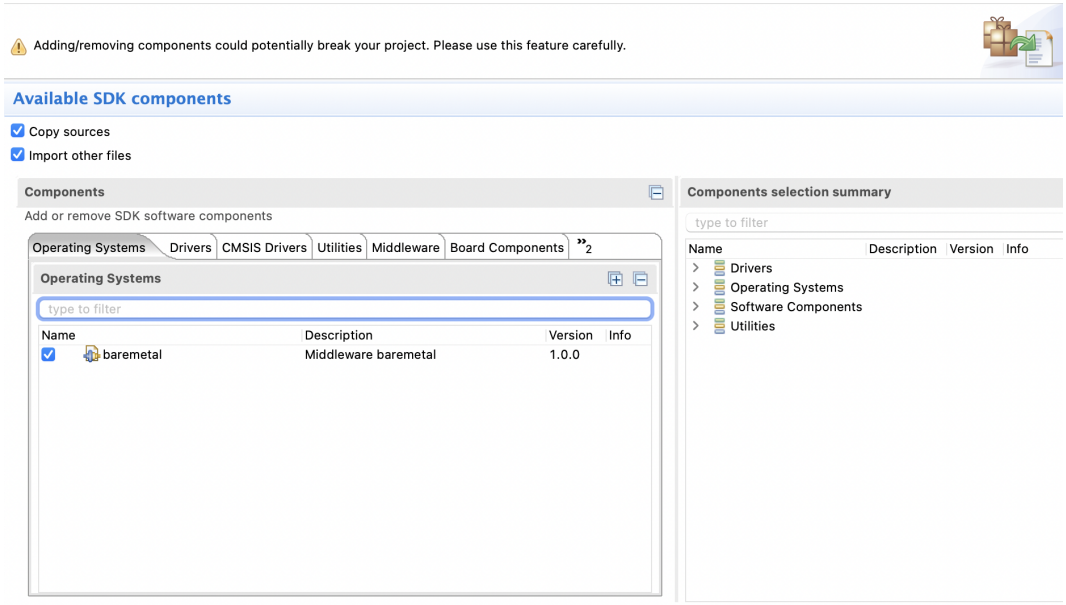
```
void BOARD_BootClockRUN(void)
{
    CLOCK_SetSimSafeDivs();

    CLOCK_SetInternalRefClkConfig(kMCG_IrclkEnable, kMCG_IrcFast, 2);
    CLOCK_CONFIG_SetFllExtRefDiv(0);
    CLOCK_SetExternalRefClkConfig(kMCG_OscselIrc);

    CLOCK_SetSimConfig(&simConfig_BOARD_BootClockRUN);
    SystemCoreClock = BOARD_BOOTCLOCKRUN_CORE_CLOCK;
}
```

   Understanding the details of this code is not mandatory; however, if you are interested in these details you may read Chapters 6 and 27, and check the difference between the FMUK66 and FRDMK66f schematics; FRDMK66f schematics are uploaded at this Link.

5. Add FTM driver to your project. In order to add new drivers from the SDK, right-click on the project name, and from "SDK Management" choose "Manage SDK Components". A pop-up window appears as shown in the following figure. From Drivers tab, check in front of ftm and press Ok.



6. In the beginning of your source file, add following line to include FTM driver.

```
#include "fsl_ftm.h"
```

7. Create pwm_setup() function and write the following code.

```
1  void pwm_setup()
2  {
3      ftm_config_t ftmInfo;
4      ftm_chnl_pwm_signal_param_t ftmParam;
5
6      ftmParam.chnlNumber = kFTM_Chnl_1;
7      ftmParam.level = kFTM_HighTrue;
8      ftmParam.dutyCyclePercent = 0;
9      ftmParam.firstEdgeDelayPercent = 0U;
10     ftmParam.enableComplementary = false;
11     ftmParam.enableDeadtime = false;
12
13     FTM_GetDefaultConfig(&ftmInfo);
14
15     FTM_Init(FTM3, &ftmInfo);
16     FTM_SetupPwm(FTM3, &ftmParam, 1U, kFTM_EdgeAlignedPwm, 5000U, CLOCK_GetFreq(
       kCLOCK_BusClk));
17     FTM_StartTimer(FTM3, kFTM_SystemClock);
18 }
```

In line 3, `ftm_config_t` is responsible for configuring the FTM and we use the default configuration for it by calling the function at line 13. The structure `ftmParam` at line 4 is used for configuring the PWM channel, and the values we assign to `ftmParam` correspond to the PWM signal connected to UI_LED_RED. After choosing the configuration value, we should initialize FTM and setup its PWM as shown in lines 15 and 16. Note that the fifth parameter of `FTM_SetupPwm` function is the PWM frequency, and it is chosen to be 5KHz but it can be changed to any other value (the frequency of PWM is conditioned

by the clock rate of FTM). Remember that our goal is to control the LED brightness, so the value of the frequency doesn't matter as long as it is higher than the range that human eye can notice the flicker.

8. In the main function add `pwm_setup()` and use `scanf()` to input the duty cycle from the console, as shown in the following code.

```
1  int main(void)
2  {
3      char ch;
4      int duty_cycle = 0;
5
6      /* Init board hardware. */
7      BOARD_InitBootPins();
8      BOARD_InitBootClocks();
9      BOARD_InitDebugConsole();
10
11     pwm_setup();
12
13     scanf("%d", &duty_cycle);
14
15     FTM_UpdatePwmDutycycle(FTM3, kFTM_Chnl_1, kFTM_EdgeAlignedPwm, duty_cycle);
16     FTM_SetSoftwareTrigger(FTM3, true);
17
18     while (1)
19     {
20     }
21 }
```

The duty cycle is updated internally by using the function at line 15, and to propagate this update to the working FTM a software trigger should be issued as in line 16 (To read more about PWM software synchronization trigger, check Section 45.5.11.2). Note that the range of the duty cycle is from 0 to 100, and it is recommended not to add scanf along with the duty cycle updating logic inside the while loop. The reason for that is the overhead coming from Semihost logic which tampers the timing of FTM. In the next Lab, we will use a better communication between the computer and the board which does not affect the timing of the other modules.

9. Run the code and enter different values for the duty cycle and notice the change in the LED brightness.

Problem 5

Write a code similar to the one in Experiment 4, but instead of driving one UI_LED, drive the three LEDs simultaneously. Accordingly, the input from the console will be RGB hex-code. For example, if the input is FFFF00, UI_LED_RED and UI_LED_GREEN will light with 100% intensity while UI_LED_BLUE will be off and the resultant color will be yellow.