EECS 388 Lab #2

C Programming

In this lab, we will first review necessary C programming background for embedded system development. You will then review the blinky code used in the lab 1 and modify it to blink all red/green/blue color leds in sequence.

Part 1: C Programming Background

(Credit: Part 1 is partially based on material developed by Prof. Lothar Thiele, ETHZ)

Number system.

A number can be represented in many different number systems. The three most common number systems you need to know are Binary, Decimal, and Hexadecimal.

Binary system uses a combination of two symbols (0 and 1) to represent a number. Decimal uses 10 symbols (0,1, … ,9) and Hexadecimal uses 16 symbols (0,...,9,a,...,f). The number of symbols of a number system is called its base number.

	Symbols	Base
Binary	0,1	2
Decimal	0,1,2,3,4,5,6,7,8,9	10
Hexadecimal	0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f	16

For example, if a base number is ‘b’, then the number “123” with the base ‘b’ can be interpreted as follows.

$123_b = 1*b^2 + 2*b^1 + 3*b^0$

You can replace ‘b’ with 2 in binary, 10 in decimal, and 16 in hexadecimal. All work the same way.

***Quiz**: convert the following numbers into decimal*

101_2 =
1111_2 =
11_{16} =
$1f_{16}$ =

In C code, you can use any of these number systems as shown below.

```
/* same number in different number systems */
int val = 26;           // decimal
int val = 0x1a;         // hexadecimal
int val = 0b00011010;   // binary
```

In embedded systems programming, hexadecimal numbers are very commonly used because they are easier to convert to binary numbers. The reason is that each hexadecimal symbol can represent 16 different values (0 - 15), which can be represented by 4 binary symbols. For example, the hexadecimal number 0x1a can be converted to a binary number as follows:

0x	1	a
0b	0001	1010

This conversion is much easier than what is necessary for decimal to binary conversion. Because ultimately binary is what the computer internally uses, using hexadecimal numbers is often very convenient when you directly interact with the computer hardware.

Data types

C language provides four basic data types: `char, int, float, double`. Each type can be combined with modifiers: `signed, unsigned, short, long`. For example, `unsigned char` type is unsigned character type. Unfortunately, the number of bits for each data type can vary depending on the processor architecture. On a 32 bit architecture, like the RISC-V processor in our HiFive1 board, for example, the `int` type (integers) has the storage size of 32 bit, while on a 64 bit architecture, e.g., your PC, the same `int` type has the storage size of 64 bit.

The following table is the basic data type information for your HiFive 1 Rev B board.

	size	range	
	(bits)	signed	unsigned
`char`	8	[-128, +127]	[0, 255]
`short int`	16	[-32768,+32767]	[0, 65535]
`int`	32	[-2147483648, +2147483647]	[0, 4294967295]

If you are unsure of your architecture or want to write portable code, it is often a good practice to use fixed-width integer data types as follows.

`int8_t, int16_t, int32_t`	signed 8 bit, 16 bit, and 32 bit integers, respectively
`uint8_t, uint16_t, uint32_t`	unsigned 8 bit, 16 bit, and 32 bit integers, respectively

Note that you must include `stdint.h` in your code to use these fixed-width integer types.

```
#include <stdint.h>
uint8_t var8;
uint32_t var32;
```

***Quiz**: What will be the output of the following code. And why?*

```
uint8_t var = 256;
printf("%d\n", var);
```

Endian

The endianness refers to the order in which bytes are stored in memory. In little-endian format, the least significant byte (LSB) is stored first (i.e., in lower memory address), while in big-endian format, the most significant byte (MSB) is stored first. For example, let's suppose a 4 byte integer value of 0x12345678 is stored in your memory, starting at address 0x1000, the memory layout will look as follows:

Memory address	little-endian	big-endian
0x1000	0x78	0x12
0x1001	0x56	0x34
0x1002	0x34	0x56
0x1003	0x12	0x78

Your target board uses little-endian format (so does your PC and most ARM based embedded systems).

Basic operators

There are many basic operators that you should know in order to understand and write C code efficiently.

Arithmetic: +, -, *, /, %
Conditional: == (equal), > (greater), < (smaller), >= (greater or equal), <= (smaller or equal),
Logical operators: && (AND), || (OR), ! (NOT).
Bitwise operators: & (AND), | (OR), ^ (XOR), ~ (complement)

Note that while logical operators operate on whole numbers and return a binary (0 or 1), bitwise operators operate on each bit individually and return an integer. See the following examples.

Bitwise AND operation:	`0b1100 & 0b1001 = 0b1000`
Bitwise OR operation:	`0b1100 | 0b1001 = 0b1101`
Bitwise complement operation:	`~(0b1100) = 0b0011`

Shift: << (shift left), >> (shift right)

Right shift by 2: 0b1100 >> 2 = 0b0011
Left shift by 1: 0b1100 << 1 = 0b1000

Assignment operators: += , -=, *=, /=, %=

example	equivalent expression
a += b	a = a + b
a -= b	a = a - b

Now, let's look at a real code example, which is defined in eecs388_lib.c of l1-blinky project.

```
void gpio_write(int gpio, int state)
{
 uint32_t val = *(volatile uint32_t *) (GPIO_CTRL_ADDR + GPIO_OUTPUT_VAL);
 if (state == ON)
   val |= (1<<gpio);
 else
   val &= (~(1<<gpio));
 *(volatile uint32_t *) (GPIO_CTRL_ADDR + GPIO_OUTPUT_VAL) = val;
 return;
}
```

This function is used to assign ON (1) or OFF (0) value to a specific gpio pin. The function first loads the current value of the GPIO output register into the 'val' variable (32bit unsigned integer). Then, it updates a single bit of the variable, which corresponds to the specified gpio pin, and writes the entire 32bit value back to the GPIO output register.

In this code, you can see how assignment operators (|=, &=) and shift operators (<<) and negation operator (~) work. Answer the following quizzes.

```
   val |= (1<<gpio);
```

***Quiz**: suppose `val = 0, gpio = 3`, what is the value of 'val' after the operation? Answer in both binary and hexadecimal number representations.*

```
   val &= (~(1<<gpio));
```

***Quiz**: suppose `val = 8, gpio = 3`, what is the value of 'val' after the operation? Answer in both binary and hexadecimal number representations.*

Part 2: RGB Blinky

Download the sample project as follows.

```
$ cd ~/Documents/PlatformIO
$ wget https://ittc.ku.edu/~heechul/courses/eecs388/l2-blinky_rgb.tar.gz
$ tar zxvf l2-blinky_rgb.tar.gz
```

Add the l2-blinky folder into VSCode workspace.
The provided code is the same as the l1-blink project in Lab 1, but includes additional information on what you need to do to blink RGB leds instead of just the green led.

Task 2.1. Review the EECS388 library

Review the EECS388 library header (`eecs388_lib.h`) and the implementation (`eecs388_lib.c`).

Task 2.2. Implement RGB blinky

Follow the instructions in the code and modify the code (`eecs388_blink.c`) to blink RGB leds.