Prof. Lothar Thiele

Embedded Systems - HS 2020

Sample solution to Lab 0

Date :   23.9.2020

Prelab − Filling the gaps

Goals of this Lab

You are expected to already be familiar with C programming. If not, your are **strongly invited** to carefully read through the *C Programming Crash Course* part of the Embedded System Companion. [1] **You will need** basic C knowledge to successfully go through the labs.

The goal of this Prelab is to cover the additional background necessary for the forthcoming Embedded Systems labs. The first part of the lab is a crash course on the following topics:

- Numeral systems
- Ordering and Endianness
- Generics types
- C operators
- Overflows

The second part of the lab is a set of short programming tasks designed to highlight the specific problems and pitfalls one may fall into when working with embedded systems.

Once you have completed these tasks, you will be good to go for the rest of the labs!

1   C programming for embedded systems

Warning 1: Basic C programming
You should already be familiar with C programming. If not, your are **strongly invited** to have a look at the *C Programming Crash Course*. [a] **You will need** basic C knowledge to successfully go through the labs.   [a] https://lectures.tik.ee.ethz.ch/es/labs/companion.pdf

Task 1: Warm-up

In programming (like for many things), it is important to be precise with the terms one uses, otherwise it is difficult to understand one another.

As a warm-up, let's have a look at the simplest C program one can write: `HelloWorld.c`

---

[1] https://lectures.tik.ee.ethz.ch/es/labs/companion.pdf

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     printf("Hello world!\n");
7     return 0;
8 }
```

Clicker Questions – #1

Which lines in this program are …

(a) processor instructions? **6 and 7**

(b) preprocessor directives? **1 and 2**

(c) function prototypes? **None**

(d) part of a function definition? **4 to 8**

Many embedded system programmers use C. Once again, you should be generally familiar with the language (**if not**, go read up the *C Programming Crash Course*! [2]). Moreover, embedded system programming also requires some "low-level" features of C which you may not know.

The rest of this section gives a quick overview of what you should know. Some clicker questions at the end will verify that you understand the main concepts.

1.1 Numeral systems

A number can be represented in different *numeral systems*, corresponding to different *bases* (*i.e.,* the number of different symbols usable). In embedded system programming, mainly three systems are used: binary, decimal, and hexadecimal.

Table 1: Main numeral systems for embedded programming

Numeral system	Base	Symbols	Interpretation of "11"
Binary	2	0 1	3
Decimal	10	0 1 2 3 4 5 6 7 8 9	11
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 a b c d e f	17

Recap 1: Bits and Symbols

The binary symbols (0 and 1) are also called *bits*.

All numeral systems work similarly. Each symbol correspond to a power of the base. The symbol's value is a multiplication factor.

For example with an arbitrary base b:

$$153_b = 3 * b^0 + 5 * b^1 + 1 * b^2$$

---
[2]<https://lectures.tik.ee.ethz.ch/es/labs/companion.pdf>

In the decimal format, when reading 153, you interpret it as $3*1 + 5*10 + 1*100$. This is base 10. It works exactly the same for the hexadecimal and binary systems.

$$
\begin{aligned}
153_{16} &= 1 * 16^2 + 5 * 16^1 + 3 * 16^0 \\
&= 1 * 256 + 5 * 16 + 3 * 1 \\
&= 339 \\
1001_2 &= 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 \\
&= 1 * 8 + 0 + 0 + 1 \\
&= 9
\end{aligned}
$$

In C, one can use any of those systems to work on numbers, as long as you inform the compiler with a "header" before the number: "0x" for hexadecimal, "0b" for binary. Without precision, all numbers are interpreted as decimal.

```
/* Equivalent definitions of the same number */
int myNumber = 339;             // myNumber in decimal
int myNumber = 0x153;           // myNumber in hexedecimal
int myNumber = 0b101010011;     // myNumber in binary
```

Why can't we just use decimal? Computers only understand 0 and 1. Therefore, binary is the "natural" numeral system of a computer. Moreover, embedded systems programmers often have to work at the "bit-level" (*e.g.,* when dealing with *registers*). In practice, you may have an 8-bit register connected to GPIO pins. To set the third pin to high, one must set the register equal to an integer value such that, in binary, the third bit equals 1. We will practice this in Task 5.

Relation between binary and hexadecimal The hexadecimal system is extremely common in programming, for a simple reason: it is compact, converts easily to binary and is easier to read.

The idea is simple. One hexadecimal symbol can take 16 different values (from 0 to 15), which is the same as a 4-symbol binary number. For example

$$
\begin{array}{ccccc}
\text{in decimal} & 4 & 13 & 6 & 15 \\
\text{0b} & 0100 & 1101 & 0110 & 1111 \\
\text{0x} & 4 & d & 6 & f
\end{array}
\quad \Rightarrow \quad \text{0b 0100 1101 0110 1111 = 0x4d6f}
$$

Thus, two hexadecimal symbols "encode" one Byte, *i.e.,* 8 bits.

1.2 Ordering consideration

Most and least significant bits The *most significant bit* (MSB) is the bit of a binary number having the greatest value (which depends on the number of bits: 2^{N-1} for a N-bit number), *i.e.,* the left-most one.
Conversely, the *least significant bit* (LSB) is the bit of a binary number having the smallest value, which is always 1, *i.e.,* the right-most one.

```
//   0b _ _ _ _ _ _ _ _
//        ^             ^
//      MSB           LSB
```

Endianness The *endianness* refers to the sequential order in which Bytes are arranged into larger numerical values when stored in memory or when transmitted (over a bus, the radio, etc.).

In *big-endian format*, the most significant Byte (*i.e.,* the Byte containing the MSB) is stored/sent first.
In *little-endian format*, the least significant Byte (*i.e.,* the Byte containing the LSB) is stored/sent first.

1.3 Generic integer types

The C programming language provides four different data types, *e.g.,* char (for characters), int (for integers), float and double (for fractionals). These types can be either signed or unsigned.

A variable can contain any number within a range that depends on the number of bits reserved in memory for this variable. For example, with 8 bits, one can store any number within $[0, 255]$ for an unsigned integer or $[-128, +127]$ for a signed one.

In C, the **number of bits used to encode integer types is not fixed**. This varies between implementations (*e.g.,* depending on the platform or the CPU). The problem is that when you declare an int variable in C, it is not clear how many bits are then reserved in memory. Thus, if you run the same program on different platforms, the size of an int can change, which may lead *e.g.,* to *overflows*.

To avoid this problem, embedded programmers use *generic integer types* (also called *fixed-width integers*). The idea is simple: the type uintN_t is used to store an unsigned integer number encoded with N bits. N can generally be 8, 16, 32, or 64. The following table summarizes these types and their value range.

Table 2: Ranges of 8, 16, and 32-bits fixed-width integer types

	Unsigned types			Signed types	
Type	Min value	Max value	Type	Min value	Max value
uint8_t	0	255	int8_t	-128	127
uint16_t	0	65535	int16_t	-32768	32767
uint32_t	0	4294967295	int32_t	-2147483648	2147483647

Recap 2: Standard Integer library

These generic types are defined in the stdint.h library. Don't forget to include it in your project, or the compiler will complain!

1.4 C Operators

The C language provides a wide range of operators that you should know about.

Arithmetic operators Addition (+) subtraction (−) multiplication (∗) division (/) and modulo (%).

Recap 3: Integer division and modulo

In "normal" calculation, $9/4 = 2.25$. However, in C, the result is 2. This is because the result of an operation on integers in also an integer. The compiler neglects the term after the decimal point.
The modulo operator (%) computes the remainder. If a = 9 is divided by b = 4, the remainder is 1 (*i.e.,* a%b = 1). The modulo operator can only be used with integers.

Increment and Decrement A short-hand notation; *e.g.,* "a = a+1" and "a++" are equivalent.

Assignment operators Another short-hand notation (see Table 3).

Tests Equal (==) Greater (>) Greater or equal (>=)
Not equal (!=) Smaller (<) Smaller or equal (<=)

Table 3: C Assignment Operators

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Logical operators Logical AND (&&), OR (||) and NOT (!)

Bitwise operators Bitwise AND (&), OR (|), XOR ($\wedge$) and complement ($\sim$) – See Snippet 1.

Warning 2: Logical vs Bitwise Operators

It is **really important** to understand the difference between logical and bitwise operators!

- A logical operator is *global* and returns a binary result (0 or 1).
- A bitwise operator operates *on each bit* individually and returns an integer (see Snippet 1).

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)


   Bitwise AND Operation of 12 and 25          Bitwise OR Operation of 12 and 25
     00001100                                    00001100
   & 00011001                                  | 00011001

     --------                                    --------
     00001000   = 8 (In decimal)                 00011101   = 29 (In decimal)


   Bitwise XOR Operation of 12 and 25          Bitwise complement Operation of 12
     00001100
   ^ 00011001                                  ~ 00001100

     --------                                    --------
     00010101   = 21 (In decimal)                11110011   = 243 (In decimal)
```

Snippet 1: Example of bitwise operators

Left- and right-shift The left- and right-shift operators (<< and >>, respectively) shifts all bits by a certain number of bits. It is rather easy to understand with examples:

```
212    = 11010100        // (In binary)
212>>2 = 00110101        // (In binary) [Right-shift by two bits]
212>>7 = 00000001        // (In binary)
212>>8 = 00000000        //
212>>0 = 11010100        // (No Shift)
```

```
212    =    11010100     // (In binary)
212<<1 =   110101000     // (In binary) [Left-shift by one bit]
212<<0 =    11010100     // (Shift by 0)
212<<4 = 110101000000    // (In binary) = 3392 (In decimal)
```

Note that, as shown in the last example, a left-shift can lead to "increase" the number of bits. However, keep in mind that this may happen only if there is "room left" in memory. Consider the following example:

```
uint8_t b = 255 ;        // b = 11111111 (In binary)
printf("%u\n", b);       // Prints: ''255''
printf("%u\n", b << 4);  // Creates a temporary variable of 12 bits and prints it.
                         // -> Prints: ''4080''

b = b << 4 ;             // Left-shifts and assigns back to b, which is 8-bit long.
                         // -> b = 11110000 (In binary)
printf("%u\n", b);       // Prints: ''240''
```

In line 6, the bits of b are left-shifted by 4 bits then assigned back to b, which is only 8-bit long. Thus, only the least significant 8 bits are stored in b and eventually printed in line 8.

Shifts are extremely useful when one needs to work on specific bits, *e.g.,* on registers, like in the following example.

```
REG = ( REG | (0x01 << 2 ));    // Sets the third bit of REG to 1
                                // without modifying any of the other bits
```

sizeof   The sizeof operator returns the size of data (constant, variables, array, structure etc) in Bytes, as illustrate below.

```
struct student{               // the structure name
    char    name[80];         // first  variable : an array of characters
    float   size;             // second variable : an decimal value
    int8_t  age;              // third  variable : an integer
};
printf("%d bytes",sizeof(struct student));  // Prints ''82 bytes''
```

1.5   Variable overflow

Standard computers possess plenty of memory and processing power. Contrary, embedded systems are generally more resource constrained. It is therefore important to carefully choose integer types: storing a counter value expected to range between 0 and 10 in a uint64_t wastes a lot of memory!

However, one must be carefull with *overflows*. A variable is said to overflow when it is assigned a value which is outside the range that its type can normally contain. For example, a uint8_t can contain any number between 0 and 255. What happens if you assign *e.g.,* 300 to such variable?

```
uint8_t a = 300;        // assign 300, even though the max value is 255
printf("%u", a);        // Prints: ''44''
```

The problem is similar to the previous example with the left-shift: 300 requires 9 bits to be written in binary. As the variable "a" is only 8-bit long, only the least significant (lower) 8 bits of 300 will be assigned to "a", that is 00101100, which is 44 in decimal (You may also note that 44 = 300%256).

Also be careful with subtractions of unsigned integers. Negative values are not defined! If you assign a negative value, the variable "wraps-around", like in the following example.

```
uint8_t a = 0;
a--;
printf("%u", a);        // Prints: ''255''
```

Clicker Questions – #2

1. How is 0b0110 written in decimal?

   (a) 10

   (b) 0d0110

   (c) 6

   (d) 42

2. How is 0b1110 written in hexadecimal?

   (a) 0x0E

   (b) 0x10

   (c) 0xE

   (d) 0xF

3. How is 0xC written in binary?

   (a) 0b1010

   (b) 0b0110

   (c) 0b1100

   (d) 0b1110

4. How is 0xD3 written in binary?

   (a) 0b1001 0011

   (b) 0b1101 0101

   (c) 0b1100 0011

   (d) 0b1101 0011

5. What is the output of the printf instruction (line 8)?

```
1  int main()
2  {
3      uint8_t counter = 10;
4      /*
5       * Some code...
6       */
7      reset(counter);
8      printf("%u", counter);
9
10     return 0;
11 }
12 void reset(uint8_t x)
13 {
14     x = 0;
15 }
```

   (a) 0

   (b) 10

   (c) 42

   (d) Impossible to know

6. What is the value of sizeof(int)?

   (a) 1

   (b) 2

   (c) 16

   (d) Impossible to know

7. What is the value of sizeof(uint16_t)?

   (a) 1

   (b) 2

   (c) 16

   (d) Impossible to know

8. Let us assume the following definition

```
1  uint16_t REG = 0xC1;
```

   Which instructions lead to REG = 0b 1100 0011?

   (a) REG = (REG | (0x01 >> 1))

   (b) REG = (REG || (0x01 >> 1))

   (c) REG = (REG | (0x01 << 1))

   (d) REG = (REG || (0x01 << 1))

9. Consider the following program and select all correct statements among the following.

```
1 int main()
2 {
3     uint8_t *ptr;
4     uint8_t x = 0;
5     ptr = &x;
6
7     printf("%u", *ptr);
8     printf("%u", &ptr);
9     printf("%u", &x);
10
11     return 0;
12 }
```

(a) Line 7 prints the address of x.

(b) Line 7 prints the value of x.

(c) Line 7 prints the address of ptr.

(d) Line 7 prints the value of prt.

(e) Line 8 prints the address of x.

(f) Line 8 prints the value of x.

(g) Line 8 prints the address of ptr.

(h) Line 8 prints the value of prt.

(i) Line 9 prints the address of x.

(j) Line 9 prints the value of x.

(k) Line 9 prints the address of ptr.

(l) Line 9 prints the value of prt.

2   Putting it to practice

It is time to put all these notions into practice! We will start using the MSP-EXP432P401R LaunchPad Development Kit which has been provided to you. For now, you just need to know how to plug it in (see Figure 1). More details about what it is/what it contains will be given in Lab 1.
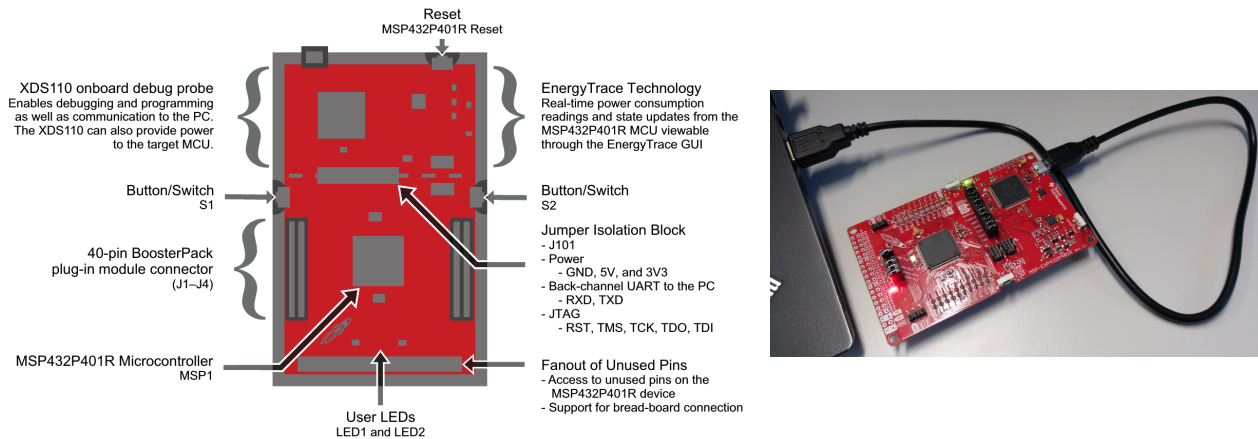
Figure 1: A basic overview of the MSP-EXP432P401R LaunchPad Development Board (left) and connected to a laptop via USB (right).

We use Code Composer Studio (CCS) as development environment. CCS can be started by typing `ccstudio` into a terminal. It allows easy integration of the LaunchPad, software, etc. Furthermore, it makes compiling of a whole project a one-click task by generating and executing the necessary makefiles, and features extensive debugging options (more details in the following labs).

Task 2: Your first embedded system program

Task 2.1: Flash your very first program!

- Download *lab0.zip* from the course website. [3]
- Open CCS and import lab0.zip to your workspace (see Figure 4 in the Appendix).
- Connect the LaunchPad to the Computer.
- Build and flash the application (, see Figure 5,6,7)
- Start the execution ( ) and observe the functionality of the program. What is the purpose of the application?

Solution for Task 2.1:
The application blinks a red LED.

Warning 3: Creating a delay within a program

In this lab, we use an empty for-loop for delaying the program. Note that this is **not a good solution** in practice. *Timers* should be used instead. We use this simple solution for now as timers will be covered later in Lab 2.

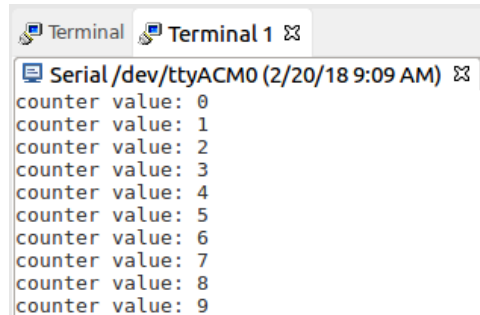[3]https://lectures.tik.ee.ethz.ch/es/labs/lab0.zip

Task 2.2: Track the number of toggling operations

We provide you with a custom `uart_println` function which prints to a terminal on your PC. Use this function exactly as you would use the standard `printf` function in C. In Task 2.2 we will modify `task_2.c` to keep track of the program execution by using a counter.

Recap 4: Print statement on embedded systems
Why do we need a custom function to print? Monitoring the execution of an embedded program is a complex task (covered in more details in Lab 2). The standard `printf` function does not help in our setting. Think about it this way: The embedded platform is lying on your desk, and executes a `printf` instruction; where would the string actually be printed?

- Stop the execution of the program (■). Open `task_2.c`.
- Declare a counter (`uint32_t` type) and initialize it to 0.
- Within the while loop, after toggling
    - Print out the counter value using the `uart_println` function.
    - Increment the counter.
- Build and debug your program. Open a terminal in CCS to see your prints (see Figure 9, 10 and 11) then start the program execution. Observe the print statements in the terminal. It should look similar to Figure 2.

Figure 2: Expected output of Task 2.2.

Solution for Task 2.2:

```
0    // Declare a toggling counter
1    uint32_t counter = 0;
2
3    while (1)
4    {
5      // Toggle an LED
6      toggle(LED_RED);
7
8      uart_println("counter value: %u", counter);
9      counter += 1;
10
11     // Delay of ~166ms
12     for (i = 0; i < 50000; i++);
13   }
```
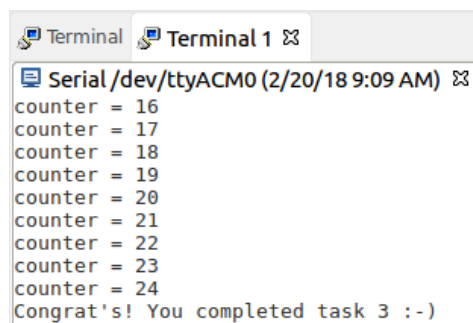
Snippet 2: Solution of Task 2.2.

Task 3: Adaptive toggling delay

We now modify the value of the delay between successive togglings, until 25 toggling operations have been executed. The program then prints a "task completed" message.

- In `main.c`, comment out `task_2` and uncomment `task_3`. Open `task_3` and observe the code provided.
- Build, flash and execute the application.
- Observe that the program never executes the "task completed" print statement. What is the problem?
- Correct the code to obtain the desired behavior. It should look similar to Figure 3.
- What is the effect of defining the `delay` variable as `static`? What would happen if we don't?

Figure 3: Expected output of Task 3.

Solution for Task 3:

Initially, the size of the delay counter `i` is too small to store the expected value. A 16-bit integer is limited to 65535. On the 22nd increment of the delay, the value reaches 66000: the variable `i` overflows! The program gets stuck in the for-loop. To solve the problem, increase the size of the `i` variable to 32 bits.

In the while-loop, having the `delay` variable defined as `static` makes it persistent: it will be defined only at the first iteration of the while-loop. Without the `static` keyword, `delay` would be reset to 0 on every loop iteration, thus leading to a constant delay of ~10ms between two toggling.

Of course, the `delay` variable could also be defined outside of the while-loop, or even as a global variable (in `main.c`)... but this will have an impact on the scope of the variable (see the *Embedded System Companion* for details).

```
0  void task_3(void)
1  {
2    // Declare a delay counter
3    volatile uint32_t i;
```

Snippet 3: Solution of Task 3.

Task 4: Use your own function

In Task 4, we encapsulate the increment and print of the `counter` value in a function called `print_and_increment`. Like in Task 3, the program must eventually exit the while loop and print the "task completed" message.

- In `main.c`, comment out `task_3` and uncomment `task_4`. Open `task_4` and observe the code provided.

- The prototype of the `print_and_increment` function is given (see line 40). Complete the definition of the function (starts at line 77) to implement the increment and print of the input argument (called `value`).

- Call your function in the while-loop at the placeholder. Rebuild, run, and observe the "task completed" print statement. The output must be similar as for Task 3 (see Figure 3).

- In line 40, replace your function prototype by the following

```
1  void print_and_increment(uint32_t *value);
```

  Rewrite your function implementation and the call in the while-loop such that the program behavior remains the same.

- What is the difference between the two function techniques? How are they called?

Solution for Task 4:

In Task 4, we manipulate two techniques related to functions: the *call by value* and the *call by reference*, which uses pointers.

The main difference is that when calling by value, the function takes as input a *copy* of the parameter's value. Thus, the function must return the updated value if we want to use it in the rest of the program (see the second cases in Snippet 4).

When using the call by reference, the function parameter is the *address* of the variable, which allows to directly modify the variable's value in memory. Therefore, there is no need to return the updated value.

The call by reference is an extremely useful and common technique. You must get comfortable using it!

The function definitions:

```
0  #if callByReference
1  void print_and_increment(uint32_t *value)
2  {
3      // Print counter value and increment it
4      uart_println("counter = %u",*value);
5      (*value) += 1;
6  }
7  #else //call by value
8  uint32_t print_and_increment(uint32_t value)
9  {
10      // Print counter value and increment it
11      uart_println("counter = %u",value);
12      value += 1;
13      return value;
14  }
15  #endif
```

The function calls in the while-loop:

```
0      #if callByReference
1          print_and_increment(&counter);
2      #else //call by value
3          counter = print_and_increment(counter);
4      #endif
```

Snippet 4: Solution of Task 4.

Task 5: Taking control of the LEDs

In Task 5 the LEDs which are toggling will be changed. We will do this by using an 8-bit unsigned integer called `activeLED`. The MSP432 Lauchpad has 2 LEDs. One red LED (which we have been using so far) and one RGB LED. An RGB LED has three inputs which can be activated separately to get red, green, blue or any mix of these three colours.

Setting bits of `activeLED` high and low controls which of the red LED and the RGB LED inputs toggles. The mapping to the bits of `activeLED` is described below.

```
1  /*
2   * Define activeLED
3   *
4   * activeLED : (MSB) _____  _____  _____  _____  _____  _____  _____  _____ (LSB)
5   *                   **     unused bits     **   RGB_BLUE  RGB_GREEN  RGB_RED  LED_RED
6   *
7   * Set the bits in activeLED to toggle the corresponding LEDs
8   */
9  //---------------led1----------------
```

Task 5.1: Static setup of the toggling LEDs

- In `main.c`, comment out `task_4` and uncomment `task_5`. Open `task_5` and observe the code provided.

- Build, flash and execute the application. What happens?

- Modify the value of `activeLED` to toggle both the red LED *and* the red RGB LED simultaneously. Try to do this using four different ways:
  - Using a decimal number,   *e.g.,* `activeLED = 65;`
  - Using a hexadecimal number,   *e.g.,* `activeLED = 0x0001;`
  - Using a binary number,   *e.g.,* `activeLED = 0b01101010;`
  - Using predified macros,   *e.g.,* `activeLED = LED_RED;` (Hint: use bitwise operators)

- Modify the value of `activeLED` to toggle all the LEDs at once (the RGB LED should appear white).

- What LEDs will toggle if you set `activeLED` to 137? To 257? Try to predict then verify.

Solution for Task 5.1:

At the beginning, only the red LED toggles. To toggle both the red LED and the red RGB LED, one must set the least two bits of `activeLED` to one (given how `activeLED` is encoded).

Similarly, to have all LEDs toggling, the least four bits of `activeLED` must be set to one.

```
0   /* activeLED definitions that toggles both LED_RED and RGB_RED */
1     //   uint8_t activeLED = 3;                        // Using decimal number
2     //   uint8_t activeLED = 0x03;                     // Using an hexadecimal number
3     //   uint8_t activeLED = 0b00000011;               // Using a binary number
4     //   uint8_t activeLED = LED_RED | RGB_RED;        // Using a macro
5
6   /* activeLED definitions that toggles all LEDs */
7     //   uint8_t activeLED = 15;                        // Using decimal number
8     //   uint8_t activeLED = 0x0f;                      // Using an hexadecimal number
9     //   uint8_t activeLED = 0b00001111;               // Using a binary number
10    //   uint8_t activeLED = LED_RED | RGB_RED | RGB_GREEN | RGB_BLUE;    // Using a macro
```

Snippet 5: Solution of Task 5.1.

To understand which LED would toggle if you set `activeLED` to a given value, one must look at this value written in binary form.

137 = 0b0 1000 1001 $\Rightarrow$ red LED and blue RGB LED toggle
257 = 0b1 0000 0001 $\Rightarrow$ only the red LED toggles

Task 5.2: Dynamic LED toggling

In this final task, the goal is to dynamically change the toggling LEDs depending on the value of the `counter` according to the specification in Table 4. For example, if `counter = 10`, 10%3=1 thus the LED_RED and RGB_GREEN should toggle.

Table 4: Specification of the expected toggling pattern of the LEDs.

(counter % 3) =	0	1	2
LED_RED	1	1	1
RGB_RED	1	0	0
RGB_GREEN	0	1	0
RGB_BLUE	0	0	1

Modify the code in `task_5.c` (where indicated by the placeholders) to execute the desired toggling pattern. Try doing this in two different ways:

- Using conditions (*e.g.,* `if` statements).
- Without conditions (*e.g.,* using shift operators).

Solution for Task 5.2:

Using if statements is simple to understand but more expensive / longer to execute for the target. Each statement is evaluated in turn by the processor, then the corresponding instruction is executed. Using shift operators, we can cut this down to only one instruction.

There are many ways of solving this task without using if statements. We just give here one example.

```
uint8_t activeLED = 0;

while(1)
{
    /* using if statements */
    if(counter%3 == 0 ){
        activeLED = LED_RED | RGB_RED;
    }
    else if(counter%3 == 1){
        activeLED = LED_RED | RGB_GREEN;
    }
    else if(counter%3 == 2){
        activeLED = LED_RED | RGB_BLUE;
    }

    /* using shift operators */
    activeLED = LED_RED | (1 << (counter%3 + 1) );
```

Snippet 6: Solution of Task 5.2.

Congrat's! If you managed all these tasks, you should now be 100% ready for the upcoming labs :-)

Appendix
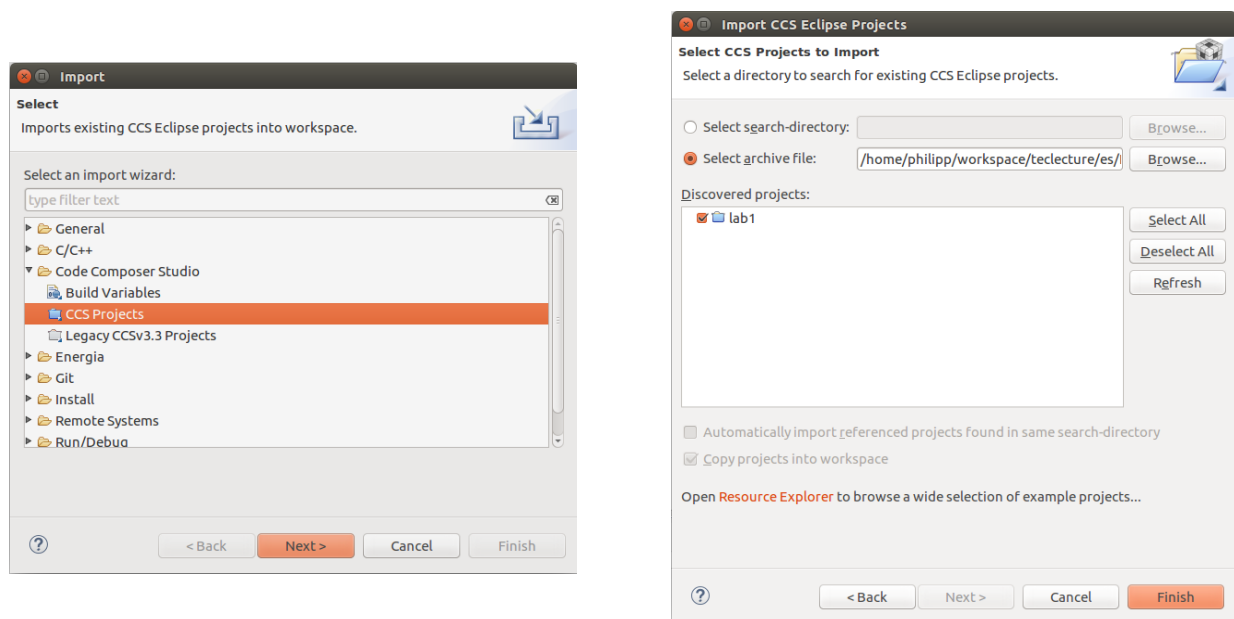
Figure 4: How to import a CCS project into your workspace in the CCS using the Import Wizard (File→Import). If the source code should be copied to the workspace, the corresponding check mark has to be set.
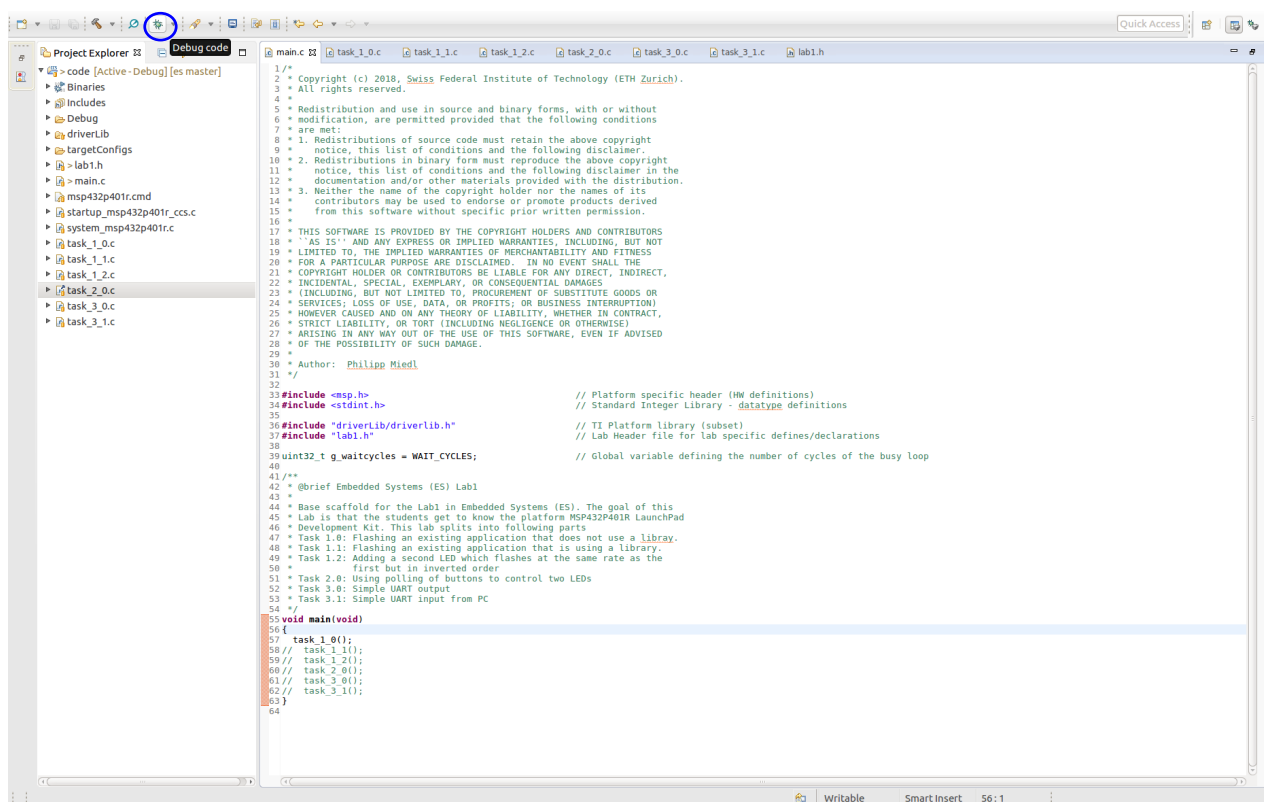

Figure 5: The standard view of the CCS. The button to build and start debugging a project is marked with a blue circle.
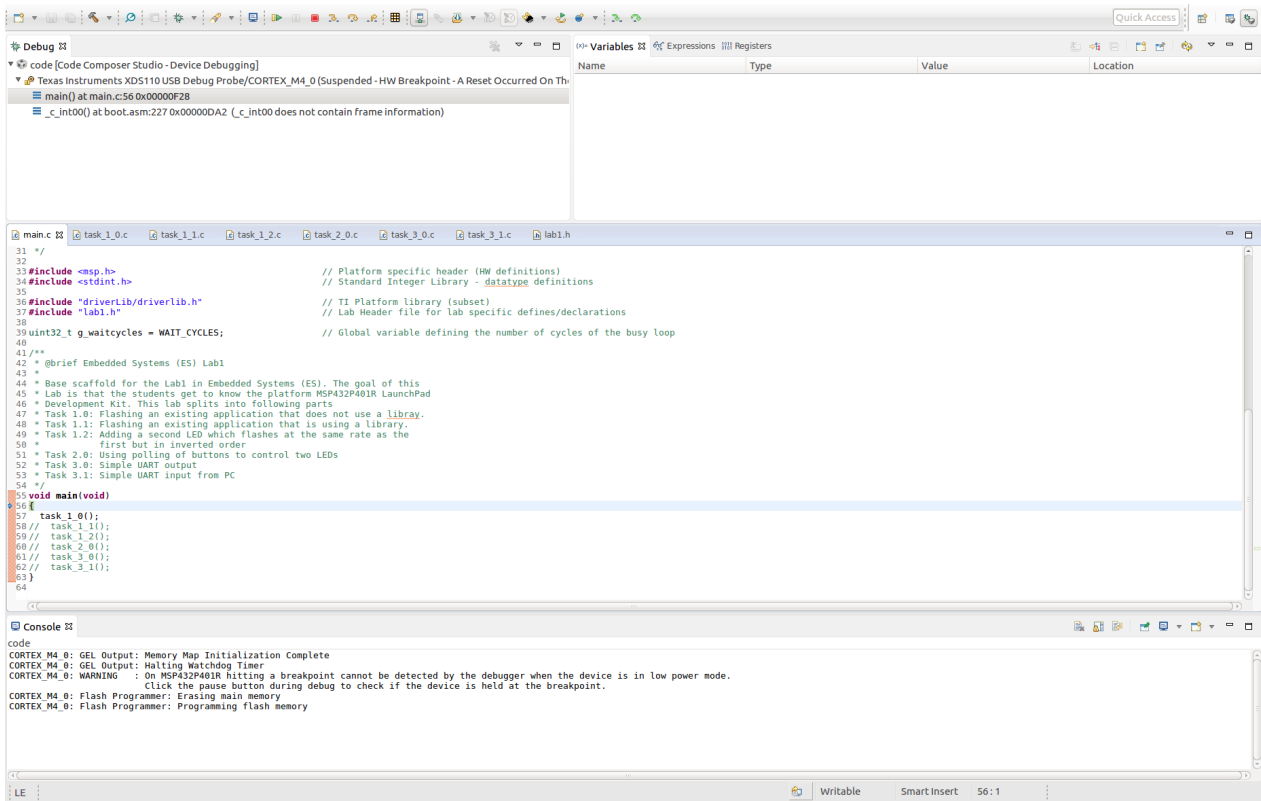
Figure 6: The debug view of the CCS.

Figure 7: Control buttons for debugging in the CCS Debug View. From left to right: Start/Continue Execution, Pause Execution, Terminate Execution, Step In, Step Over, Step Out.
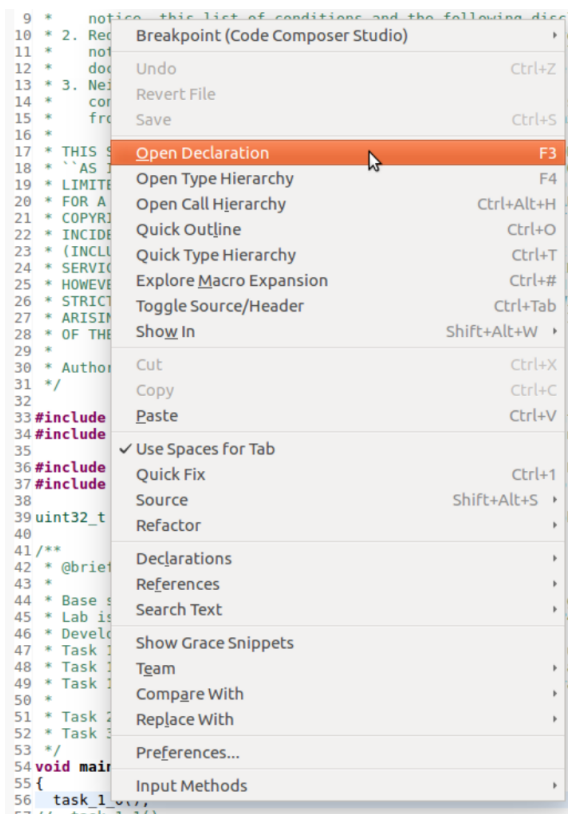
Figure 8: The context menu in CCS allows easy code browsing with searches (Declarations, References, Search Text) or other commands (i.e. Open Declaration). It opens by right-clicking a function or a variable. Alternatively, one can also use the sequence *Ctrl+Left Click*.
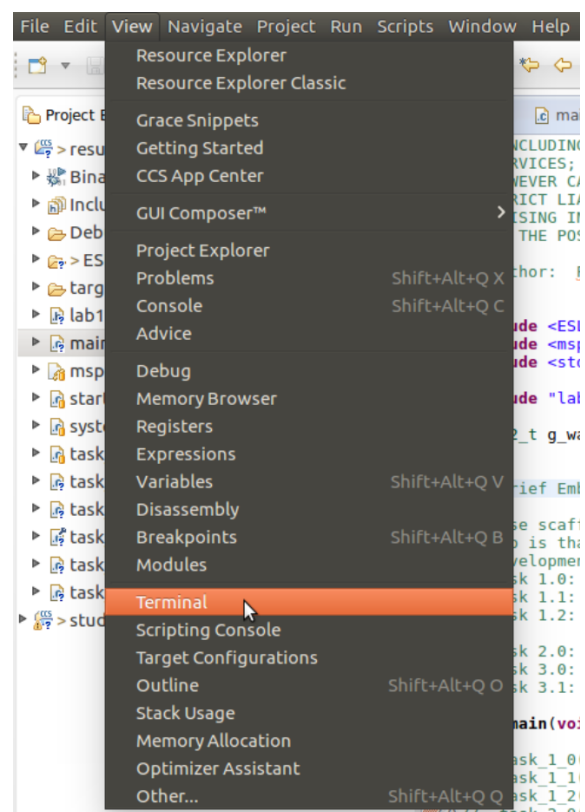

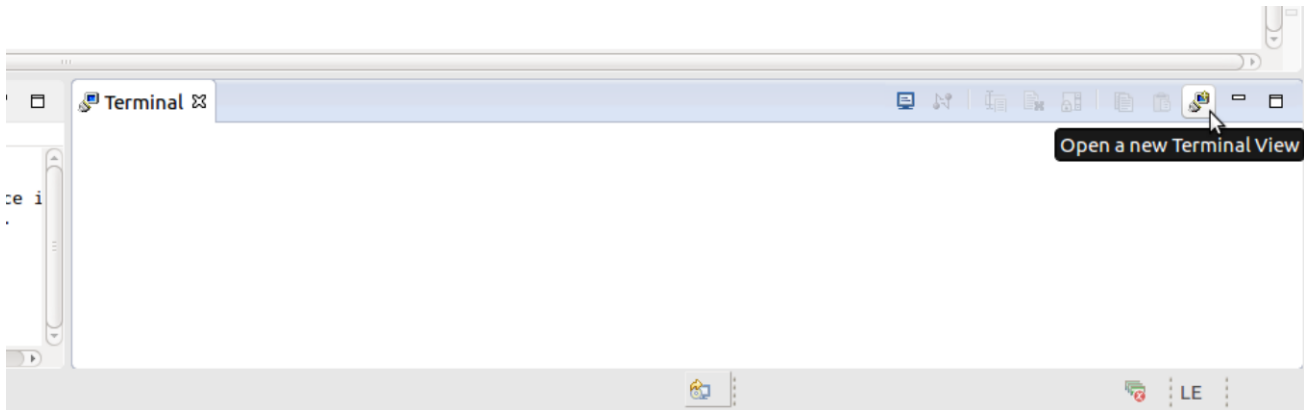**Figure 9:** To open the terminal view, go to View→Terminal.

Figure 10: The terminal view opens in the lower right corner where a new terminal view can be opened.
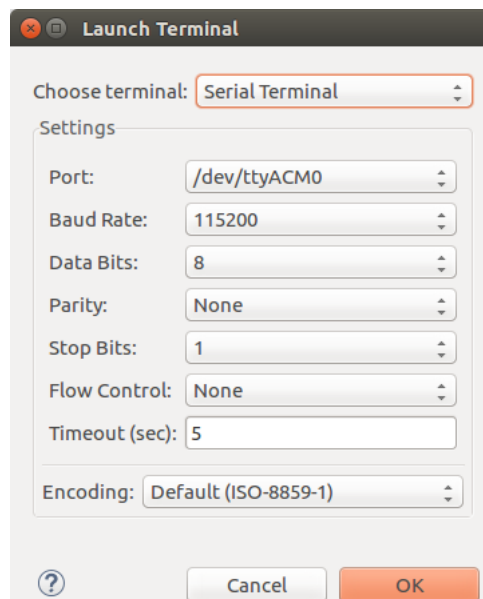

Figure 11: The menu opened after pressing the new terminal button allows the configuration of the serial connection.