Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze

Prof. Lothar Thiele

Embedded Systems - HS 2020

Sample solution to Lab 1

Date : 30.9.2020

LaunchPad Basic Bare-Metal Programming

Goals of this Lab

- Get to know the MSP-EXP432P401R
- Learn how to use registers for configuration
- Get to know and use library functions
- Implement simple GPIO with peripherals
- Understand and implement polling
- Establish a simple UART communication

Introduction

In this lab session, we get to know the development platform MSP-EXP432P401R LaunchPad and the Code Composer Studio (CCS), which are used throughout all ES labs. We will also familiarize ourself with concepts like direct register access, polling and serial communication. The following introduction should give a broad overview of the working environment, but it is not necessary to comprehend all the details to complete this laboratory.

The Workspace - Structure and IDE . As a development environment the CCS is used. 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.

CCS Usage The code for the labs can be downloaded from the course website https://www.tec.ee.ethz.ch/education/lectures/embedded-systems.html as zip-file. Figure 6 illustrates how to directly import an existing CCS project from a zip-file into your workspace. Figure 7, 8 and 9 illustrate the most important controls for running and debugging an application. If you are navigating a big project, the context menu allows you to easily find function or variables using the search features of CCS (see Figure 10). Alternatively, you can also use the combination *Ctrl+Left Click* onto a function or variable name to go to its definition. In addition, CCS includes a terminal (Figure 11, 12 and 13) and several consoles, which give extensive debug and error output. Please *always* carefully read compile warnings and error messages as they are designed to make development easier and help to identify errors or issues in the code. The features presented here are just a small fraction of the full functionality of the CCS.

MSP432P401 The MSP432P401 is a low power mixed signal microprocessor featuring a 32-Bit ARM Cortex M4F CPU. It contains several different memories (flash main memory, information memory, SRAM and ROM) and has the Texas Instruments (TI) MSP432P401 peripherals driver libraries installed. Furthermore, it offers flexible clocking (tune-able clock sources), several timing units (timers, Pulse Width

Modulation (PWM), capture and compare, ...) and serial communication interfaces (Universal Asynchronous Receiver Transmitter (UART), I2C, Serial Peripheral Bus (SPI),...). The MSP432P401 also includes an Analogue to Digital Converter (ADC) unit, analogue comparators and various Input/Output (I/O) pins.

MSP-EXP432P401R LaunchPad Development Kit The MSP-EXP432P401R LaunchPad Development Kit is an evaluation module based on the *TI MSP432P401* microprocessor. It features simple peripherals (buttons, LEDs, etc.) and the XDS110-ET debug probe. The XDS110-ET debug probe allows direct programming of the target device (in our case the MSP432P401) without the use of an external

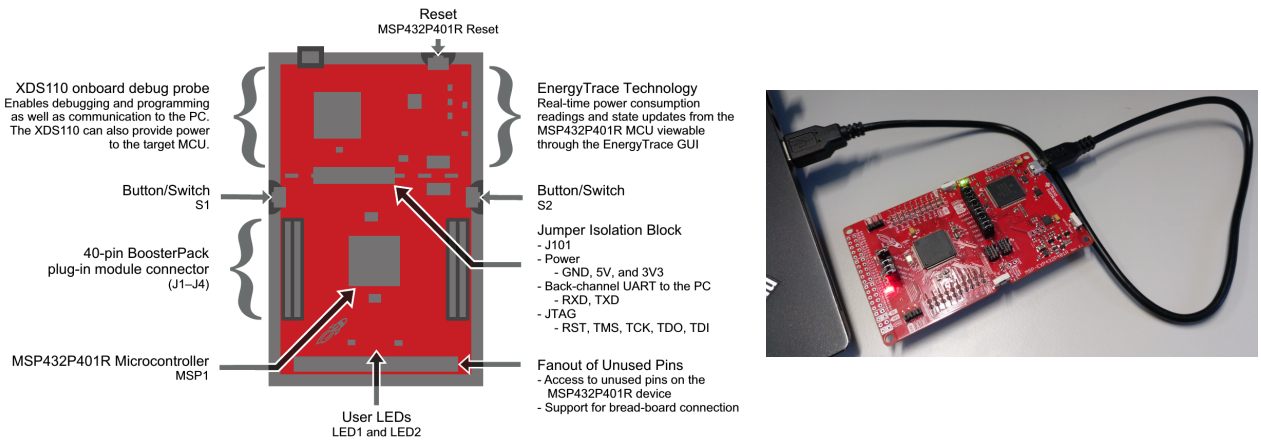

Figure 1: A basic overview of the MSP-EXP432P401R LaunchPad Development Board (left) and connected to a laptop via USB (right).

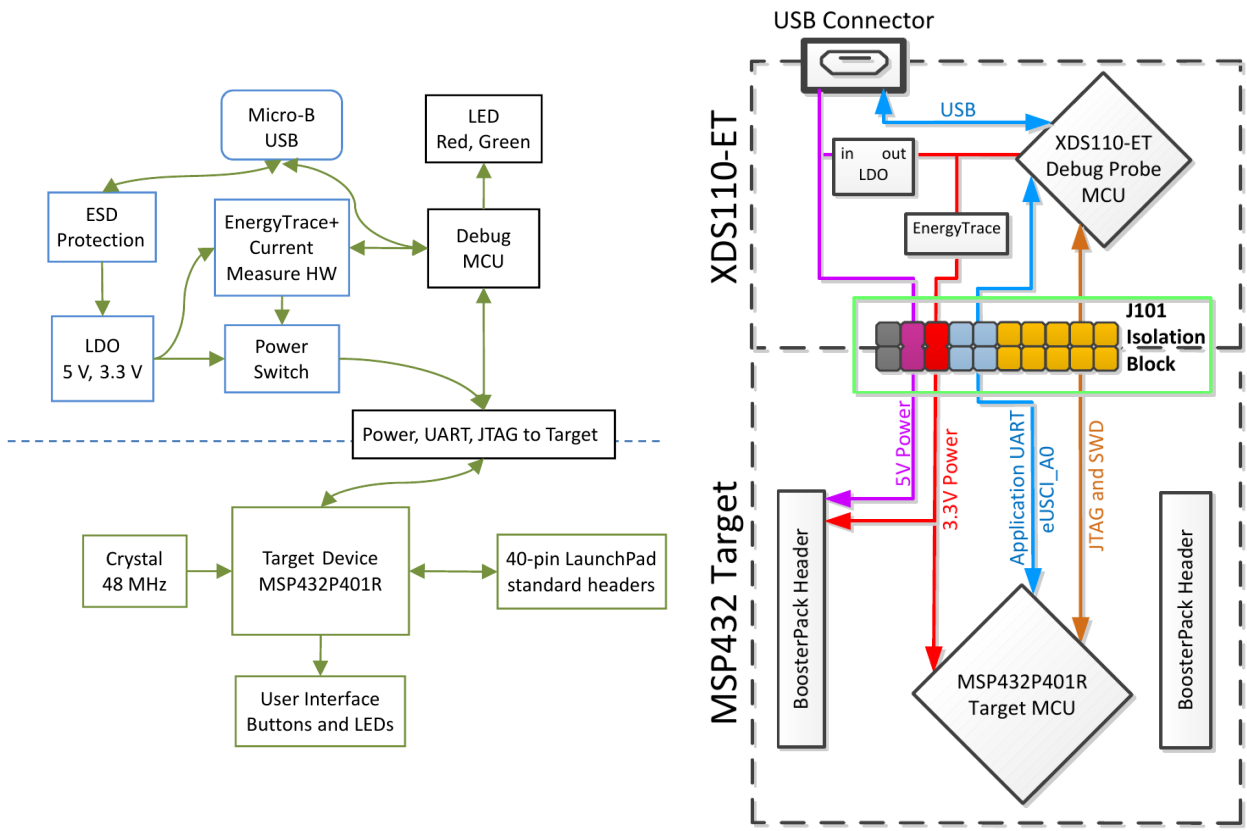

Figure 2: Block diagrams of the MSP-EXP432P401R LaunchPad Development Board. A general block diagram (left) and one highlighting the XDS110-ET isolation block and the debug components (right).

programming device and also gives the opportunity to debug an application and direct communication with the PC. The block diagrams in Figure 2 illustrate the basic structure of the MSP-EXP432P401R. All debug and communication signals run through the XDS110-ET debug probe, therefore there is *never* a direct connection between the target and the computer. A direct connection between the computer and the MSP432P401 is only possible using an external setup. Furthermore, for high precision tests, it is possible to isolate the target through the J101 Isolation block (jumpers).

Embedded System Documentation The most important documents of an embedded system are, among others, Datasheets, Users Guides, Technical Reference Manuals, Application Notes, Erratas or Schematics. These documents are available for...

- the microprocessor at the manufacturers website, e. g. TI (http://www.ti.com/),
- components of the microprocessor, e. g. ARM microprocessors IP[1] documentation,
- hardware boards, e. g. the TI MSP-EXP432P401R LaunchPad,
- extension peripherals, e. g. the TI BoostXL Sensors booster pack, and its components, for example sensors like the Bosch BMM150 Geomagnetic Sensor,
- as well as software, e. g. the TI MSP432P401 Peripheral Driver Library, or
- complementary information in wikis or online fora

Therefore, every embedded system comes with a large number of documentation files. For this lab and all following labs, all the necessary files are provided on the course website https://lectures.tik.ee.ethz.ch/es/labs//lab_documents.zip as zip-file. The documentation is needed to determine

- the pins peripherals are connected to,
- register addresses and memory mapped peripherals,
- configuration values for modules,
- bit masks and their meaning,
- functionality of modules (e. g. serial communication modules, ...)
- ...

Therefore, often it is essential to have access to all parts of the documentation to be able to use the functionality of an embedded system to its fullest extent.

Bare-Metal Programming Programming bare-metal means to program an embedded system without the use of an underlying Operating System (OS) like Linux, FreeRTOS or similar.

Watchdog Timer The watchdog timer is an internal timer unit of an embedded system that will reset the system if it times out. Under normal operation, the system will reset the watchdog timer regularly to prevent it from timing out, but in case of an error, the watchdog timer will reset the device. This is used to prevent a deployed embedded system which cannot be accessed easily (e.g. Mars Rover) from being stuck if an error occurs. In laboratory exercises, the watchdog timer is disabled as the embedded system is easily accessible.

Defines, Macros and in-line Functions Preprocessor directives like defines and macros, as well as in-line functions, are often used when programming embedded systems to make the code more readable and easier to maintain. Examples for a define, macro and in-line function are given in Snippet 1 and 2. The `#define` directive, used for defines and macros, is a preprocessor directive that instructs the compiler to replace certain code parts as defined. In contrast to that, the in-line function is just a request to the

[1] IP is (in this case) short for Intellectual Property and refers to a ready to use component of a System-on-Chip (SoC), for example a microprocessor, that is provided by a third party vendor (e.g. ARM) and can be integrated by chip manufacturers (e.g. TI).

compiler. This means, depending on the optimization configuration of the compiler, function calls to in-line functions can be replaced with the function body or are treated like normal function calls. However, in contrast to macros, in-line functions are subject to strict parameter checking and are therefore considered to be "safer".

```
#define GPIO_PIN0                                                    (0x0001)
```

Snippet 1: Example for a define.

```
inline void lab1_configureUART(
      const eUSCI_UART_Config *config
    )
{
  // Selecting P1.2 and P1.3 in UART
  // mode
GPIO_setAsPeripheralModuleFunctionOutputPin
             (
               GPIO_PORT_P1,
               GPIO_PIN2 | GPIO_PIN3,
               GPIO_PRIMARY_MODULE_FUNCTION
             );
  // Configuring UART Module
  UART_initModule(UART_INTERFACE, config);
  // Enable UART module
  UART_enableModule(UART_INTERFACE);
}
```

```
#define MACRO_lab1_configureUART(          \
      config                               \
    )                                      \
{                                          \
  /* Selecting P1.2 and P1.3 in UART    */ \
  /* mode                               */ \
GPIO_setAsPeripheralModuleFunctionOutputPin\
             (                             \
               GPIO_PORT_P1,               \
               GPIO_PIN2 | GPIO_PIN3,      \
               GPIO_PRIMARY_MODULE_FUNCTION \
             );                            \
  /* Configuring UART Module            */ \
  UART_initModule(EUSCI_A0_BASE, config);  \
  /* Enable UART module                 */ \
  UART_enableModule(EUSCI_A0_BASE);     */ \
}
```

Snippet 2: Examples for an in-line function (left) and a macro (right).

Clicker Questions

1. To which General Purpose Input Output (GPIO) Port are the *LED1* and *LED2* connected? (Hint: Check the Schematic Pages 1 and 2)

   (a) Port 1 & Port 3
   (c) Port 1 & Port 2
   (b) Port 2 & Port 4
   (d) Port 3 & Port 4

2. Using `C`, which of the following operation allows toggling of the LSB of an 8-bit integer X?

   (a) `X ^ 0x01`
   (c) X | $0x01$
   (b) X & $0x01$
   (d) X && $0x01$

3. What is the declaration of the function `I2C_initSlave()`? (Hint: DriverLib Users Guide Section $12.6.3$)

   (a)
   ```
   I2C_initSlave(
       uint16_t moduleInstance, uint_16_t slaveAddress,
       uint_8_t slaveAddressOffset, uint32_t slaveOwnAddressEnable
   )
   ```
   (b)
   ```
   void I2C_initSlave(
       uint32_t slaveOwnAddressEnable, uint_fast16_t slaveAddress,
       uint_fast8_t slaveAddressOffset, uint32_t moduleInstance
   )
   ```

(c) `void I2C_initSlave(`
`uint_fast16_t slaveAddress, uint_fast8_t slaveAddressOffset,`
`uint_fast32_t moduleInstance, uint_fast32_t slaveOwnAddressEnable`
`)`

(d) void I2C_initSlave(
uint32_t moduleInstance, uint_fast16_t slaveAddress,
uint_fast8_t slaveAddressOffset, uint32_t slaveOwnAddressEnable
)

4. What is the UART module baud rate divider (BRDIV) for a baud rate of 4800 at low frequency baud rate generation, using the default setting of SMCLK as Clock Source? (Hint: LaunchPad Users Guide Section 2, http://processors.wiki.ti.com/index.php/USCI_UART_Baud_Rate_Gen_Mode_Selection)

   (a) 312
   (b) 625
   (c) 39
   (d) 156

5. Which of the following statements is *not* zero, when for the pins 5 *or* 7 of Port 2 the primary or tertiary module function is enabled? (Hint: Datasheet Table 6-1 & Technical Reference Manual Chapter 10 on PxSEL0 and PxSEL1)

   (a) `HWREG8(0x40004C00 + 0x0C) & 0x75`
   (b) `HWREG8(0x40004800 + 0x2B) & 0xA0`
   (c) HWREG8(0x40004C00 + 0x0B) & 0xA0
   (d) `HWREG8(0x40004800 + 0x4A) & 0x75`

   (a) First, check how the control bits PxSEL1 and PxSEL0 have to be set in order to configure the primary or teritary module function. This can be found in the Technical Reference Manual on page 501 in table $10-2$. Here we can see that PxSEL0 is set (*non-zero*) in both cases.
   (b) Now use the Datasheet to identify the base address range of the port modules and find the reference to table $6-21$. In table $6-21$ you can identify that the base address is 0x40004C00 and the offset for P2SEL0 is 0x0B.
   (c) Switching back to the Technical Reference Manual, we can use section $10.4.7$ on page 519 to identify the bitmask for the register read out as $0xA0$, as we only want to consider the bits P2SEL0.5 and P2SEL0.7.

Task 1: Flashing, Library Usage and Simple I/O

Using the CCS we can easily establish the connection to a MSP-EXP432P401R connected via USB and program it. In this task you will learn how to flash a functioning application using GPIO functions onto the chip. Furthermore, you will learn how to alter and extend the application using library functions.

Recap 1: General Purpose Input Output (GPIO)
The GPIO pins can be configured either as input or output. GPIO pins can be used to drive peripherals (i.e. LEDs, small actuators, switches) or read inputs (i.e. sensors, buttons).

Task 1.1: Flashing an Application

- Download *lab1.zip* from the course website.
- Open the CCS and import lab1.zip to your workspace (see Figure 6).
- Connect the LaunchPad to the computer.
- Build and flash the application using the `Debug` button. Any warnings can be ignored at this point as they are only relevant in Task 3. Note that the CCS is changing to the debug view.
- `Start` the execution and observe the functionality of the program.
- What is the purpose of the application?

Solution for Task 1.1:
The application lets LED 1 blink.

Task 1.2: Using Library Functions instead of Hard-Coded Register Access

In this task we want to use library functions instead of any hard-coded register access. Please follow the instructions below:

```
#define REGBASEADR  ((uint32_t)(0x40004C00))  // Base addr. of Port 1 configuration register
#define REGOFS_SEL0 ((uint32_t)(0x0000000A))  // Addr. Offset for Select0-Register in Port 1
#define REGOFS_SEL1 ((uint32_t)(0x0000000C))  // Select1 offset in Port 1  conf. reg.
#define REGOFS_DIR  ((uint32_t)(0x00000004))  // Direction offset in Port 1 conf. reg.
#define REGOFS_OUTV ((uint32_t)(0x00000002))  // Output Value offset in Port 1 conf. reg.
```

Snippet 3: Defines used in Task 1.1 which cannot be used anymore in Task 1.2.

Recap 2: Register Access

When programming embedded systems it is possible to directly access special registers to allow low level hardware configurations. This can either be done through macros using addresses (see Snippet 10) or predefined structs (see Snippet 11). In many cases, manufacturers provide so called Hardware Abstraction Layer (HAL) or drivers to make register access easier when doing bare-metal programming or using an OS.

- Create a file called `task_1_2.c` and copy the code from `task_1_1.c` to `task_1_2.c`. Change the function name in `task_1_2.c` from `task_1_1()` to `task_1_2()`.
- Change the function call in `main()` to call `task_1_2()`. A declaration of the function is already provided in `lab1.h`.
- Add the include for the `ESLab1driverLib` (`#include "ESLab1driverLib/driverlib.h"`) in `task_1_2.c` before the inlude for `lab1.h` (`#include "lab1.h"`)[2].
- Open the `gpio.h` and `gpio.c` files in the `ESLab1driverLib` directory. In these files you can search for the corresponding library functions and defines to replace the direct register access commands (`HWREG16()`) in `task_1_2.c` with the right library function calls. The function `task_1_2()` has to have exactly the same functionality as `task_1_1()`, without using any of the defines used in Task 1.1 (see Snippet 3) or direct calls to the `HWREGXX` macros from `task_1_1()`.
- What is the advantage of using library functions and defines instead of direct register access?

[2]Header files somethimes have to be included in a specific order. In our case, this is due to the use of the preprocessor directive `__DRIVERLIB__H_` in `lab1.h`.

Solution for Task 1.2:
The use of library functions and defines has multiple advantages:

- The code gets more readable.
- The code is easier to maintain.
- By using driver functions the same application can be run on different platforms without changing the application code but only exchanging the drivers (given the driver interfaces are the same).

Notice that the order in which you add the includes matters. If the include of the driverLib is added after the `lab1.h`, some parts of `lab1.h` will not be active due to a processor directive. So always consider in which order you add your includes. The solution code can be found in Snippet 4.

```c
#include <msp.h>                                // Platform specific header (HW definitions)
#include <stdint.h>                             // Standard Integer - data type definitions
#include <stdio.h>                              // Standard In-/Output definitions/functions

#include "ESLab1driverLib/driverlib.h"          // TI Platform library (subset)
#include "lab1.h"                               // Lab specific defines/declarations

void task_1_2(void)
{
  WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;   // Stopping the Watchdog Timer

  uint32_t count = 0;                           // Simple counter variable
  GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0); // Set P1.0 to output direction

  while(1)
  {
    GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);  // Toggle LED1
    // ------------------------------------------------------------------------------- //
    //                                   Placeholder 1                                 //
    // ------------------------------------------------------------------------------- //
    for(count = 0; count < g_waitcycles; count++)   // Busy Loop for Delay
    {
      // ----------------------------------------------------------------------------- //
      //                                   Placeholder 2                               //
      // ----------------------------------------------------------------------------- //
    }
  }
}
```

Snippet 4: Solution code for Task 1.2

Task 1.3: Adding a blinking LED

Now we want to use LED2, which is a RGB LED. This means, in contrast to *LED1*, the RGB LED has three inputs which can be activated separately to get red, green, blue or any mix of these three colours. Please follow the instructions below:

- Create a file called `task_1_3.c` and copy the code from `task_1_2.c` to `task_1_3.c`. Change the function name in `task_1_3.c` from `task_1_2()` to `task_1_3()`.
- Change the function call in `main()` to call `task_1_3()`. A declaration of the function is already provided in `lab1.h`.
- Identify the ports and pins, the RGB LED is connected to, in the LaunchPad schematics.
- Configure the pin connected to red of the RGB LED as output.
- Implement the blinking, such that the red of the RGB LED blinks alternating to *LED1*.
- Do you need to set any register specifically to make sure the LEDs blink alternating in any case, when using the toggle output pin function?

Solution for Task 1.3:
See Snippet 5.

```c
#include <msp.h>                                // Platform specific header (HW definitions)
#include <stdint.h>                             // Standard Integer - data type definitions
#include <stdio.h>                              // Standard In-/Output definitions/functions

#include "ESLab1driverLib/driverlib.h"          // TI Platform library (subset)
#include "lab1.h"                               // Lab specific defines/declarations

void task_1_3(void)
{
  WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;         // Stopping the Watchdog Timer

  uint32_t count = 0;                                 // Simple counter variable
  GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);       // Set P1.0 to output direction
  GPIO_setAsOutputPin(GPIO_PORT_P2, GPIO_PIN0);       // Set P2.0 to output direction

  GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0);   // Make sure LED1 starts on
  GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN0);    // Make sure LED2 Red starts off

  while(1)
  {
    GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);  // Toggle LED1
    GPIO_toggleOutputOnPin(GPIO_PORT_P2, GPIO_PIN0);  // Toggle LED2 Red
    // ------------------------------------------------------------------------------- //
    //                                   Placeholder 1                                 //
    // ------------------------------------------------------------------------------- //
    for(count = 0; count < g_waitcycles; count++)   // Busy Loop for Delay
    {
      // ----------------------------------------------------------------------------- //
      //                                   Placeholder 2                                 //
      // ----------------------------------------------------------------------------- //
    }
  }
}
```

Snippet 5: Solution code for Task 1.3

Setting one LED with `GPIO_setOutputHighOnPin` and one with `GPIO_setOutputLowOnPin` makes sure that the two LED have different starting states. The LEDs may also blink alternating without this initialisation, but only if you are lucky with the initial state of the registers. Therefore it is important to initialise all the registers properly to ensure deterministic behaviour of your program, as embedded systems often do not intialise all their registers to a defined state at startup.

Task 2: GPIO Pins as Inputs with Polling

GPIO pins can also be used for input. In this task, we want to configure multiple GPIO inputs to enable user interaction using the two buttons *S1* and *S2* on the MSP-EXP432P401R. In order to avoid a so

Recap 3: Polling

Continuously sampling a certain information (i. e. register or input) is called polling. This can be used to assess the status of a peripheral or internal status informations or simply read information.

called *floating pin*, pull-up or pull-down resistors are used. A pin is called floating when there is no fixed voltage level connected to the pin, which can lead to unexpected behaviour if the pin is read. In Figure 3 the two concepts of pull-up and pull-down resistors are illustrated. A pull-up resistor will make sure the pin is on high voltage level when the button is not pressed (open circuit), and on low if the button is pressed (short circuit). The pull-down resistor works vice versa.

Task 2.1: Identifying the GPIO Configuration

- Check the LaunchPad schematics and identify the ports and pins the buttons *S1* and *S2* are connected to.
- Find out whether to use Pull-Up, Pull-Down or no pulling resistor (Hint: Schematic).

Solution for Task 2.1:
Figure 4 shows the schematics of the buttons and the connected ports. To be able to read the button status, **pull-up** resistors have to be used. If the button is not pressed there is no connection between `GND` and the port. This means that the resistance between the port and `GND` is considered to be $\infty\,\Omega$, therefore much higher than the pull-up. The whole voltage is now on the switch and the voltage level of the pin is high.

In case of a pressed button, the connection between the port and `GND` is shorted (restance $0\,\Omega$). This means that the voltage level at the pin is equal to `GND`, low, as the whole voltage is on the pulling resistor.

Task 2.2: Implement Button Polling

- Create a file called `task_2_2.c` and copy the code from `task_1_3.c` to `task_2_2.c`. Change the function name in `task_2_2.c` from `task_1_3()` to `task_2_2()`.
- Change the function call in `main()` to call `task_2_2()`. A declaration of the function is already provided in `lab1.h`.
- Configure the pins connected to the buttons *S1* and *S2* as inputs with the approriate pull resistor (up/down/no), as determined in Task 2.1.
- Use polling inside the `while`-loop (placeholder 1) to read the button status.
- If button *S1* is pressed the green RGB LED has to be on and if *S2* is pressed the blue RGB LED has to be on.
- The red LED1 and red RGB LED must still blink alternatingly.
- Can you observe any difference in terms of blinking frequency compared to Task 1.3? What about the reaction time of the buttons `S1` and `S2`?

Solution for Task 2.2:
The blinking frequency does not change notably but the reaction time of the buttons is very slow. This is due to the fact that the device is not reacting to any changes on the buttons while in the busy `for`-loop, see Snippet 6.

Task 2.3: When should we Poll?

- Create a file called `task_2_3.c` and copy the code from `task_2_2.c` to `task_2_3.c`. Change the function name in `task_2_3.c` from `task_2_2()` to `task_2_3()`.
- Change the function call in `main()` to call `task_2_3()`. A declaration of the function is already provided in `lab1.h`.
- *Move* the polling into the for-loop (placeholder 2). Rebuild, flash and start the program.

```c
#include <msp.h>                                // Platform specific header (HW definitions)
#include <stdint.h>                             // Standard Integer - data type definitions
#include <stdio.h>                              // Standard In-/Output definitions/functions

#include "ESLab1driverLib/driverlib.h"          // TI Platform library (subset)
#include "lab1.h"                               // Lab specific defines/declarations

void task_2_2(void)
{
  WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;         // Stopping the Watchdog Timer

  uint32_t count = 0;                                 // Simple counter variable
  GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);       // Set P1.0 (LED1) to output direction
  GPIO_setAsOutputPin(GPIO_PORT_P2,                   // Set P2.0, P2.1 & P2.2 (LED2) ...
                      GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2
                     );                               // ...to output direction

  GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0);   // Make sure LED1 starts in high state
  GPIO_setOutputLowOnPin(GPIO_PORT_P2,                // Make sure LED2 start...
                         GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2
                        );                            // ...in low state

  GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1,  // Set P1.1 & P1.4 (buttons S1 and...
                                       GPIO_PIN1 | GPIO_PIN4
                                      );              // ... S2) as inputs with pull-up
  while(1)
  {
    GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);  // Toggle LED1
    GPIO_toggleOutputOnPin(GPIO_PORT_P2, GPIO_PIN0);  // Toggle LED2 Red

    // ------------------------------------------------------------------------------- //
    //                                   Placeholder 1                                 //
    if(GPIO_getInputPinValue(GPIO_PORT_P1,GPIO_PIN1) == ((uint8_t)0x00))  // Poll button S1
    {
      GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);    // Set LED2 Green to high
    }
    else
    {
      GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN1);     // Set LED2 Green to low
    }
    if(GPIO_getInputPinValue(GPIO_PORT_P1,GPIO_PIN4) == ((uint8_t)0x00))  // Poll button S2
    {
      GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN2);    // Set LED2 Blue to high
    }
    else
    {
      GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN2);     // Set LED2 Blue to low
    }
    // ------------------------------------------------------------------------------- //
    for(count = 0; count < g_waitcycles; count++)
    {
      // ----------------------------------------------------------------------------- //
      //                                   Placeholder 2                               //
      // ----------------------------------------------------------------------------- //
    }
  }
}
```

Snippet 6: Solution code for Task 2.2.

- Compared to Task 1.3 and Task 2.2, which changes in terms of blinking frequency and button reaction time can you observe now?
- What are the upsides of the implementations in Task 2.2 and Task 2.3?
- Can you think of a way to implement polling such that there is no influence on the blinking frequency and the button reaction time is minimized?

Solution for Task 2.3:
The blinking frequency of the LEDs decreases drasticly. This is caused by the polling in the busy `for`-loop. In Task 2.2 one `for`-loop iteration consisted of no actions, therefore one iteration only lasted for a very short amount of time. Now, in every `for`-loop iteration the chip has to poll the buttons (access multiple registers), therefore one iteration takes much longer and the waiting time in between changing the LED states is longer (Snippet 7). While Task 2.3 does not influnce the behaviour of the LEDs, the

```c
#include <msp.h>                                // Platform specific header (HW definitions)
#include <stdint.h>                             // Standard Integer - data type definitions
#include <stdio.h>                              // Standard In-/Output definitions/functions

#include "ESLab1driverLib/driverlib.h"          // TI Platform library (subset)
#include "lab1.h"                               // Lab specific defines/declarations

void task_2_3(void)
{
  WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;         // Stopping the Watchdog Timer

  uint32_t count = 0;                                 // Simple counter variable
  GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);       // Set P1.0 (LED1) to output direction
  GPIO_setAsOutputPin(GPIO_PORT_P2,                   // Set P2.0, P2.1 & P2.2 (LED2) ...
                      GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2
                     );                               // ...to output direction

  GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0);   // Make sure LED1 starts in high state
  GPIO_setOutputLowOnPin(GPIO_PORT_P2,                // Make sure LED2 start...
                         GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2
                        );                            // ...in low state

  GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1,  // Set P1.1 & P1.4 (buttons S1 and...
                                       GPIO_PIN1 | GPIO_PIN4
                                      );              // ... S2) as inputs with pull-up

  while(1)
  {
    GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);  // Toggle LED1
    GPIO_toggleOutputOnPin(GPIO_PORT_P2, GPIO_PIN0);  // Toggle LED2 Red
    // ------------------------------------------------------------------------------- //
    //                                   Placeholder 1                                 //
    // ------------------------------------------------------------------------------- //
    for(count = 0; count < g_waitcycles; count++)
    {
      // ----------------------------------------------------------------------------- //
      //                                   Placeholder 2                                 //
      if(GPIO_getInputPinValue(GPIO_PORT_P1,GPIO_PIN1) == ((uint8_t)0x00)) // Poll button S1
      {
        GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);    // Set LED2 Green to high
      }
      else
      {
        GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN1);     // Set LED2 Green to low
      }
      if(GPIO_getInputPinValue(GPIO_PORT_P1,GPIO_PIN4) == ((uint8_t)0x00)) // Poll button S2
      {
        GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN2);    // Set LED2 Blue to high
      }
      else
      {
        GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN2);     // Set LED2 Blue to low
      }
      // ----------------------------------------------------------------------------- //
    }
  }
}
```

Snippet 7: Solution code for Task 2.3.

implementation in Task 2.3 minimizes the latency for reactions to button presses.

A possible solution only using simple loops is as follows: If the polling of the buttons is done inside the `for`-loop and the number of iterations is modified, polling can be implemented without influencing the blinking frequency of the LEDs and minimizing the button reaction time.

Task 3: Simple UART output (Optional)

Recap 4: Universal Asynchronous Receiver Transmitter (UART)

UART is an asynchronous serial communication protocol. The sender and the receiver have to agree on a transmission rate and packet format before any communication can happen. The data is transmitted in packets of 6 to 9 bits per packet. Additionally, a packet consist of a start bit, one or two stop bits and optionally parity bits (see Figure 14). Figure 15 shows the block diagram of the transmit part of one of the MSP432P401 peripheral modules capable of UART communication. It allows clock source selection, clock sub-sampling to generate the proper baud rate clock and output data buffers.

As a final task we will use UART to send a message to the Computer whenever one of the buttons is pressed.

Task 3.1: Calculating the UART Parameters

Recap 5: UART Configuration Shorthand Notation

The most common UART configuration is often given with its shorthand notation *8N1*. It defines the packet structure as follows:

(a) 8 Bits per Packet

(b) *N*o Parity bits

(c) 1 Stop bit

HINT: The calculations done in this tasks are similar to the ones in the clicker question. However, the parameters are `NOT` the same.

- Create a file called `task_3.c` and copy the code from `task_2_3.c` to `task_3.c`. Change the function name in `task_3.c` from `task_2_3()` to `task_3()`.
- Change the function call in `main()` to call `task_3()`. A declaration of the function is already provided in `lab1.h`.
- Use the datasheet to identify whether `eUSCI_A`, `eUSCI_B` or either of the two module types could be used for the UART connection. (Hint: check the eUSCI modules in the MSP432P401 Reference Manual Chapters 22 to 24)
- Determine the default setting for the `SMCLK` as clock source (Hint: MSP-EXP432P401R User Manual Chapter 2).

- Get the SMCLK frequency from the LaunchPad User Guide and calculate the UART parameters for a baud rate of 4800 using http://processors.wiki.ti.com/index.php/USCI_UART_

Baud_Rate_Gen_Mode_Selection or http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSP430BaudRateConverter/index.html.

Solution for Task 3.1:
The MSP432P401 Reference Manual Chapter 22 explains the eUSCI UART mode and states that only the eUSCI_A modules can be used for UART communication. The default setting for the SMCLK can be found in the MSP-EXP432P401R User Manual page 14 (section 2.6), which is 3 MHz. Using the provided parameters the calculator on the website can be used to determine the parameters as outlined in Figure 5.

In this scenario we can choose either the *low frequency* or the *oversampling* baud rate generation. The low frequency baud rate generation minimizes the energy consumption of the chip, while being a bit more error prone. Therefore, in case the oversampling baud rate generation is available and the energy consumption is not crucial, the oversampling baud should be chosen.

Task 3.2: Implementing UART Output

- Insert the parameters determined in Task 3.1 into the prepared struct in `lab1.h` (lines 83 to 94) and set following configuration: 8N1 with LSB first.
- Insert the necessary commands to enable the UART output. You can use the provided `lab1_configureUART()` and `uart_println` inline functions from `lab1.h` to configure and use the UART interface.
- Start the program and open a terminal in CCS to observe the UART input (see Figure 11, 13 and 12).
- Can you think of a better method to register button presses instead of polling to save resources and avoid duplicated output messages?

Solution for Task 3.2:
Instead of polling, interrupts could be used to save resources and avoid duplicated output messages. The solution code is illustrated in Snippet 8 and 9.

```
static const eUSCI_UART_Config uart_config  =
{
  EUSCI_A_UART_CLOCKSOURCE_SMCLK,                  // SMCLK Clock Source Selected
  39,                                              // Baudrate Divider BRDIV  = 39
  1,                                               // Clock Compensation Factor UCxBRF =  1
  0,                                               // Clock Compensation Factor UCxBRS =  0
  EUSCI_A_UART_NO_PARITY,                          // No Parity
  EUSCI_A_UART_LSB_FIRST,                          // LSB First
  EUSCI_A_UART_ONE_STOP_BIT,                       // One stop bit
  EUSCI_A_UART_MODE,                               // UART mode
  EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION    // Oversampling Mode
};
```

Snippet 8: UART configuration struct.

```c
#include <msp.h>                                // Platform specific header (HW definitions)
#include <stdint.h>                             // Standard Integer - data type definitions
#include <stdio.h>                              // Standard In-/Output definitions/functions

#include "ESLab1driverLib/driverlib.h"          // TI Platform library (subset)
#include "lab1.h"                               // Lab specific defines/declarations

void task_3(void)
{
  WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;         // Stopping the Watchdog Timer

  uint32_t count = 0;                                 // Simple counter variable
  GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);       // Set P1.0 (LED1) to output direction
  GPIO_setAsOutputPin(GPIO_PORT_P2,                   // Set P2.0 & P2.1 (LED2)...
                      GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2
                     );                               // ...to output direction

  GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0);   // Make sure LED1 starts in high state

  // Make sure LED2 start in low state
  GPIO_setOutputLowOnPin(GPIO_PORT_P2,
                         GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2
                        );

  GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1,  // Set P1.1 & P1.4 (buttons S1 and...
                                       GPIO_PIN1 | GPIO_PIN4
                                      );              // ... S2) as inputs with pull-up
  lab1_configureUART(&uart_config);

  while(1)
  {
    GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);  // Toggle LED1
    GPIO_toggleOutputOnPin(GPIO_PORT_P2, GPIO_PIN0);  // Toggle LED2 Red
    // ------------------------------------------------------------------------------- //
    //                                   Placeholder 1                                 //
    // ------------------------------------------------------------------------------- //
    for(count = 0; count < g_waitcycles; count++)
    {
      // ----------------------------------------------------------------------------- //
      //                                 Placeholder 2                                 //
      if(GPIO_getInputPinValue(GPIO_PORT_P1,GPIO_PIN1) == ((uint8_t)0x00)) // Poll button S1
      {
        GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);          // Set LED2 Green to high
        uart_println(str_s1);                                      // Press S1: Send str1
      }
      else
      {
        GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN1);           // Set LED2 Green to low
      }
      if(GPIO_getInputPinValue(GPIO_PORT_P1,GPIO_PIN4) == ((uint8_t)0x00)) // Poll button S2
      {
        GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN2);          // Set LED2 Blue to high
        uart_println(str_s2);                                      // Press S2: Send str2
      }
      else
      {
        GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN2);           // Set LED2 Blue to low
      }
      // ----------------------------------------------------------------------------- //
    }
  }
}
```

Snippet 9: Solution code for Task 3.2

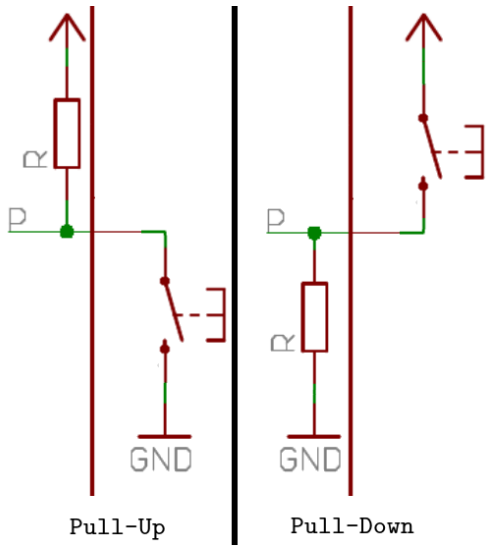

Figure 3: Schematic of a pull-up (left) and pull-down resistor.

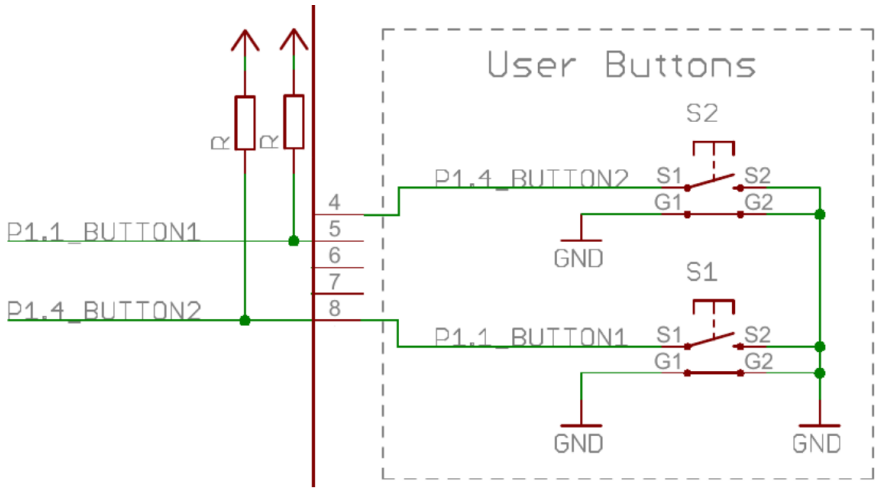

Figure 4: Schematic with enabled pull-up resistors at ports 1.1 and 1.4 to read the button status.

Clock Speed: 3000000 Baud Rate: 4800 Calculate USCI Dividers

Low Frequency Baud Rate Generation

Divider = 625; BRDIV = 625
S-Modulation UCBRSx = 0
UCxBR0UART = 113
UCxBR1UART = 2
UCxBRS = 0
UCxBRF = 0
Max. TX bit error: 0(0, 0)
Max. RX bit error (sync.: -0.5 BRCLK): -0.16000000000009063(-0.16000000000009063,0)
Max. RX bit error (sync.: +/-0 BRCLK): -0.08000000000001494(-0.08000000000001494,0)
Max. RX bit error (sync.: +0.5 BRCLK): -2.0816681711721685e-13(-2.0816681711721685e-13,1.0408340855860843e-13)
Max. RX bit error (over: -0.5,0,+0.5): 0(-0.16000000000009063,1.0408340855860843e-13)

Oversampling Baud Rate Generation

Divider= 39.0625; BRDIV= 39
S-Modulation UCBRSx= 0
F-Modulation UCBRFx= 1
UCxBR0UART = 39
UCxBR1UART = 0
UCxBRS = 0
UCxBRF = 1
Max. TX bit error: 0(0,0)
Max. RX bit error (sync.: -0.5 BRCLK): -2.0816681711721685e-13(-2.0816681711721685e-13,1.0408340855860843e-13)
Max. RX bit error (sync.: +/-0 BRCLK): 0.08000000000004531(0,0.08000000000004531)
Max. RX bit error (sync.: +0.5 BRCLK): 0.16000000000009063(0,0.16000000000009063)
Max. RX bit error (over: -0.5,0,+0.5): 0.16000000000009063(-2.0816681711721685e-13,0.16000000000009063)

Figure 5: Parameters to configure the eUSCI module to UART mode with a baud rate of 4800.

Appendix

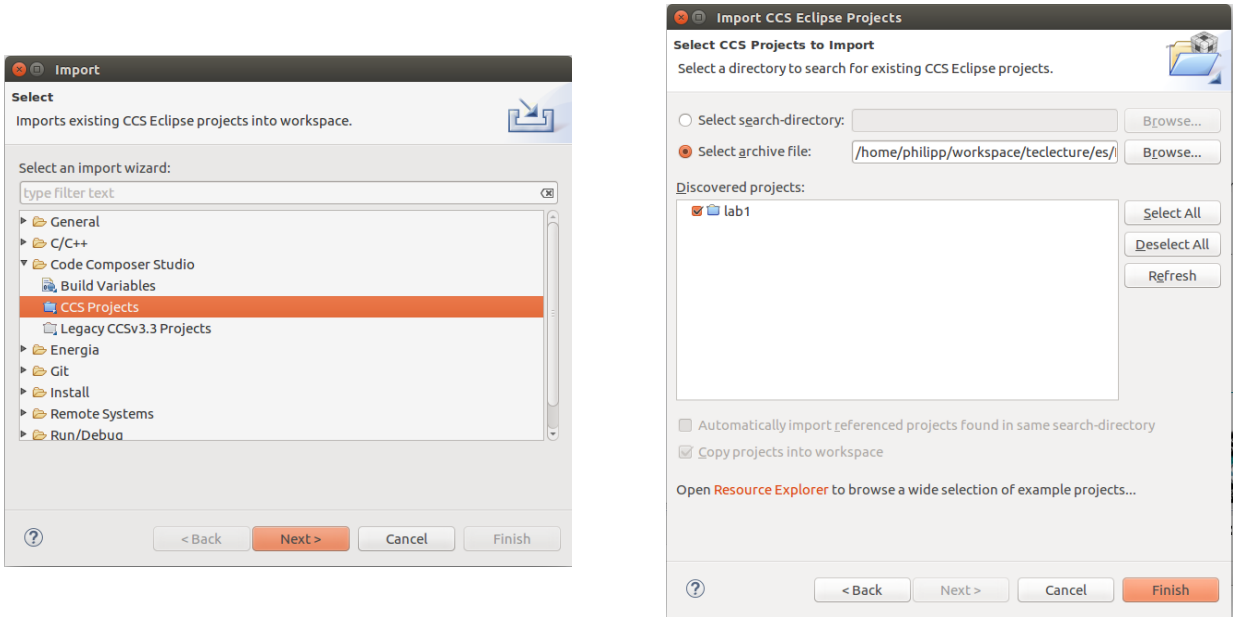

Figure 6: How to import a CCS project into your workspace in the CCS using the Import Wizard (File→Import). If the source code should be copied to the workspace, the corresponding check mark has to be set.

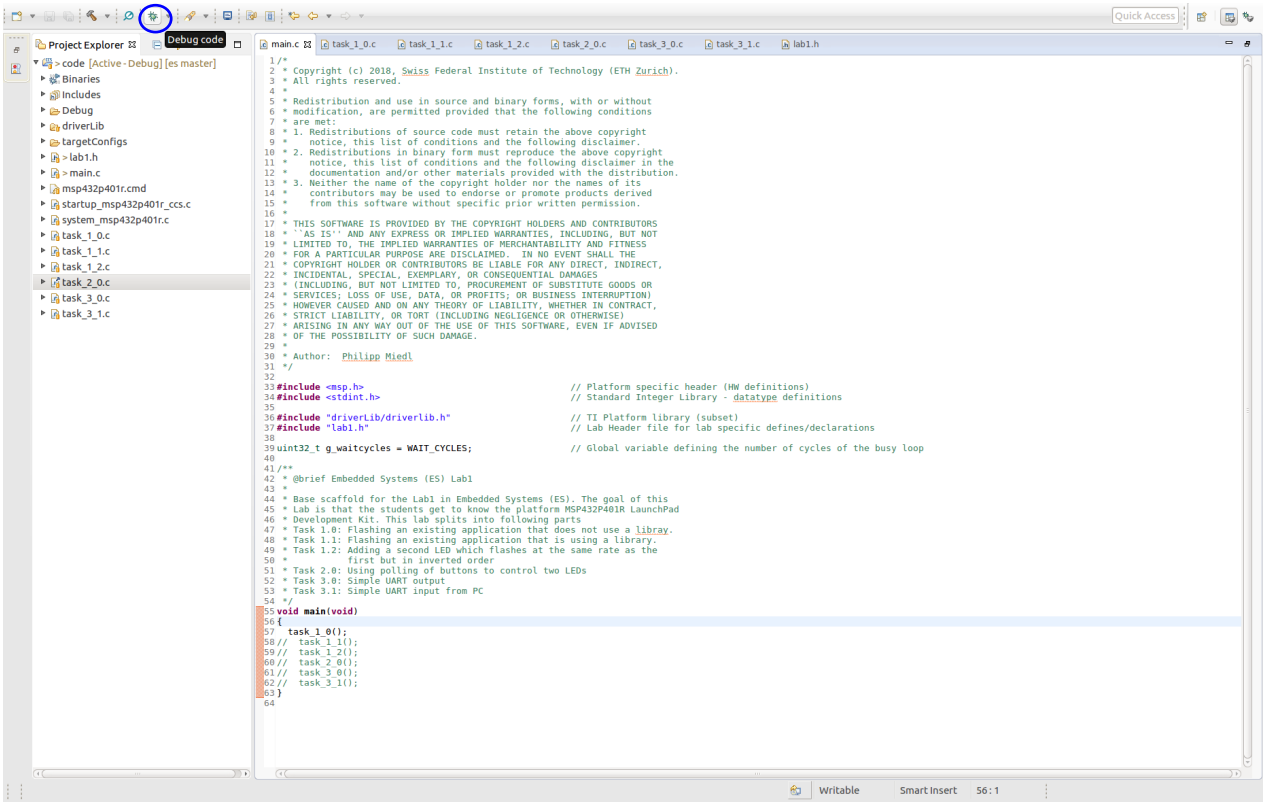

Figure 7: The standard view of the CCS. The button to build and start debugging a project is marked with a blue circle.

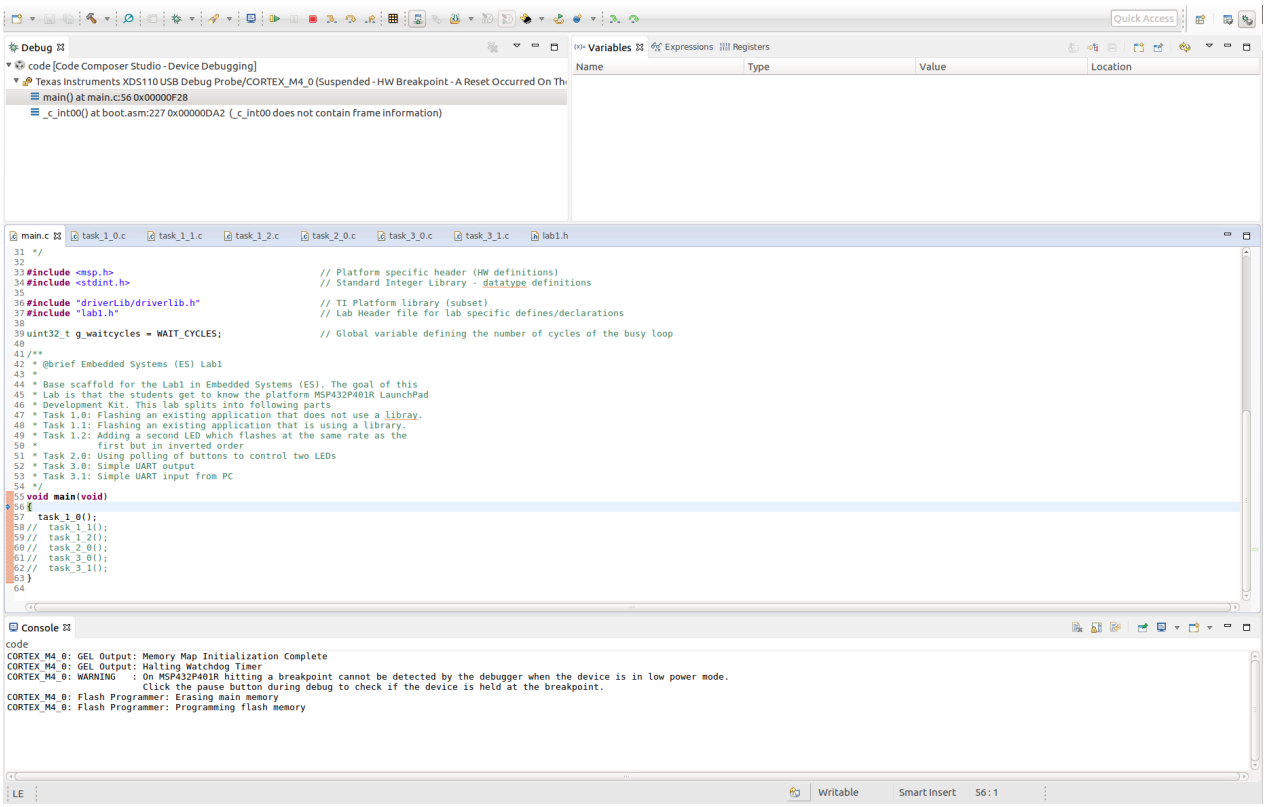

Figure 8: The debug view of the CCS.

Figure 9: 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.

```
/*****************************************************************************
* Definitions for 8/16/32-bit wide memory access                             *
*****************************************************************************/
#define HWREG8(x)          (*((volatile uint8_t *)(x)))
#define HWREG16(x)         (*((volatile uint16_t *)(x)))
#define HWREG32(x)         (*((volatile uint32_t *)(x)))
#define HWREG(x)           (HWREG16(x))
#define HWREG8_L(x)        (*((volatile uint8_t *)((uint8_t *)&x)))
#define HWREG8_H(x)        (*((volatile uint8_t *)(((uint8_t *)&x)+1)))
#define HWREG16_L(x)       (*((volatile uint16_t *)((uint16_t *)&x)))
#define HWREG16_H(x)       (*((volatile uint16_t *)(((uint16_t *)&x)+1)))
```

Snippet 10: Examples for register access macros using addresses.

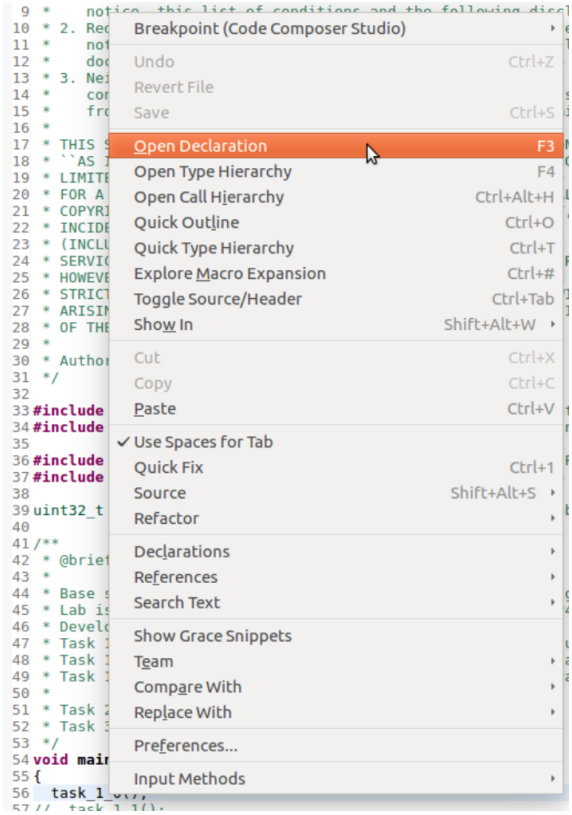

Figure 10: The context menu in CCS allows easy code browsing with searches (Declarations, References, Search Text) or other commands (i.e. Open Declaration). It opens by right-clicking a function or a variable. Alternatively, one can also use the sequence *Ctrl+Left Click*.

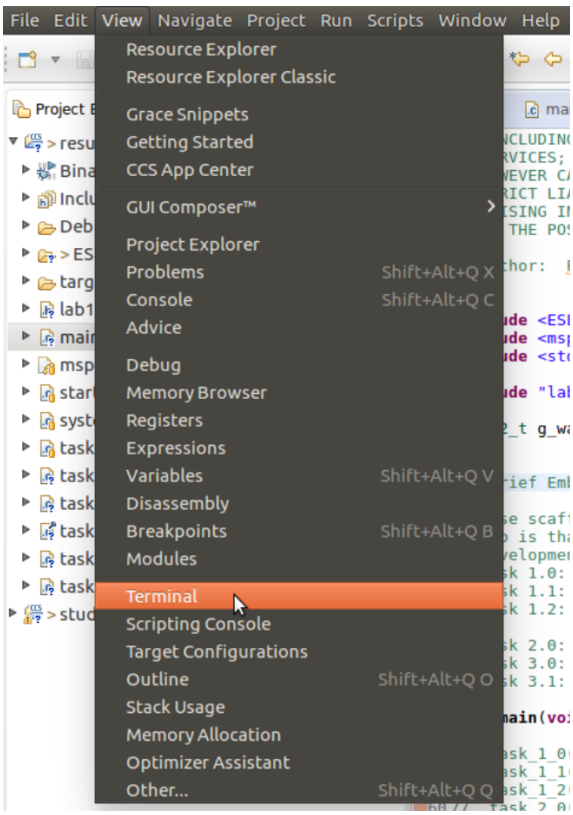

Figure 11: To open the terminal view, go to View→Terminal.

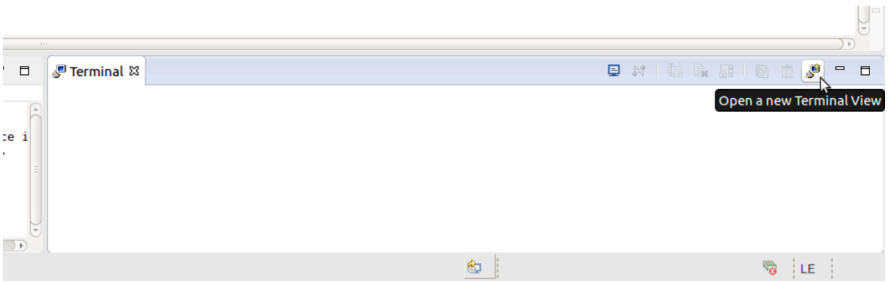

Figure 12: The terminal view opens in the lower right corner where a new terminal view can be opened.

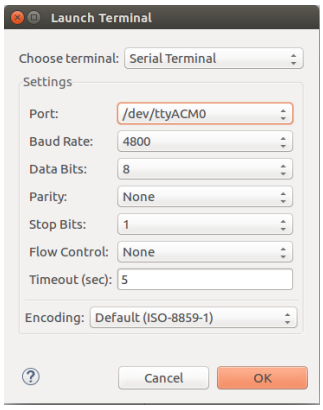

Figure 13: The menu opened after pressing the new terminal button allows the configuration of the serial connection.

```c
typedef struct {
  __IO uint16_t CTLW0;                          /*!< eUSCI_Ax Control Word Register 0 */
  __IO uint16_t CTLW1;                          /*!< eUSCI_Ax Control Word Register 1 */
       uint16_t RESERVED0;
  __IO uint16_t BRW;                            /*!< eUSCI_Ax Baud Rate Control Word
    Register */
  __IO uint16_t MCTLW;                          /*!< eUSCI_Ax Modulation Control Word
    Register */
  __IO uint16_t STATW;                          /*!< eUSCI_Ax Status Register */
  __I  uint16_t RXBUF;                          /*!< eUSCI_Ax Receive Buffer Register */
  __IO uint16_t TXBUF;                          /*!< eUSCI_Ax Transmit Buffer Register */
  __IO uint16_t ABCTL;                          /*!< eUSCI_Ax Auto Baud Rate Control
    Register */
  __IO uint16_t IRCTL;                          /*!< eUSCI_Ax IrDA Control Word Register */
       uint16_t RESERVED1[3];
  __IO uint16_t IE;                             /*!< eUSCI_Ax Interrupt Enable Register */
  __IO uint16_t IFG;                            /*!< eUSCI_Ax Interrupt Flag Register */
  __I  uint16_t IV;                             /*!< eUSCI_Ax Interrupt Vector Register */
} EUSCI_A_Type;
```

```c
    return EUSCI_A_CMSIS(moduleInstance)->RXBUF;
```

Snippet 11: Examples for an register access struct and its usage taken from the `UART_receiveData()` function.

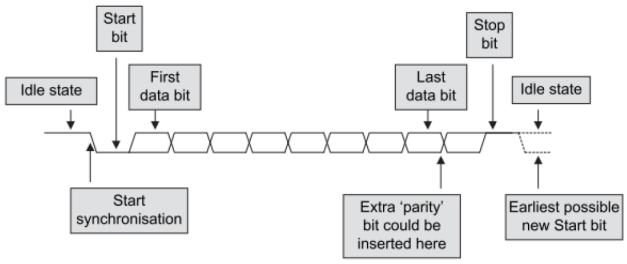

Figure 14: UART Packet Structure.

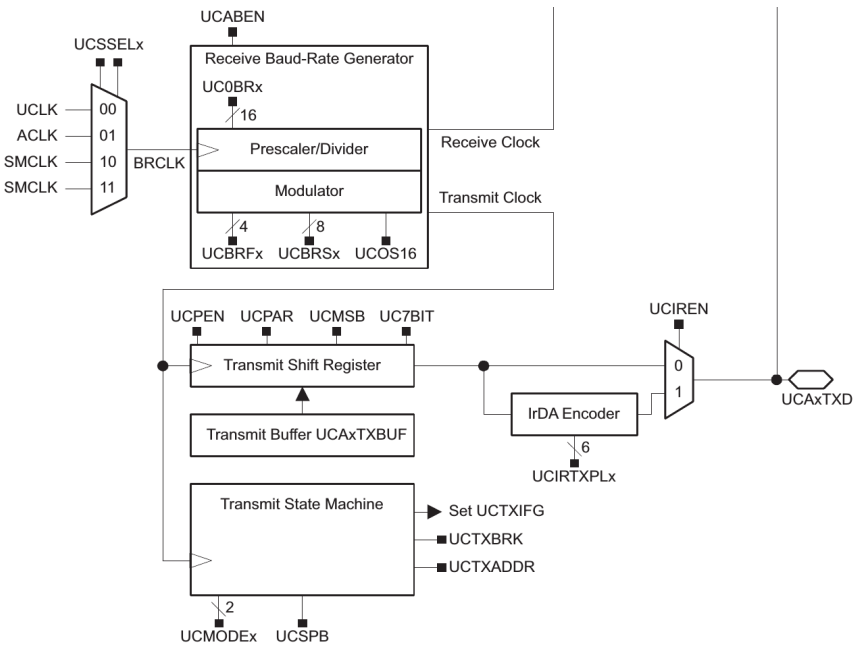

Figure 15: Block diagram of the receive part of a UART module of MSP432P401.