Digital Electronics ELEC-H-310
Introduction labs and project specifications

Contents

1 Introduction

1.1 Lab and project objectives

The objective of this project is to design an entire microcontroller system, that uses many common functionalities of a microcontroller. In this lab, we use the PSoC device from Cypress, but the skills that will be learned during this project should be applicable to different microcontroller families (DSPIC, STM32, ...).

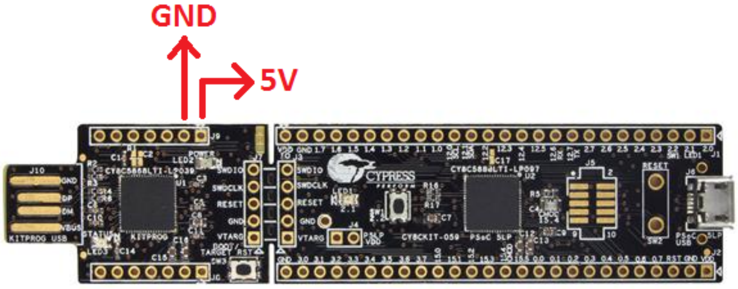


Figure 1: PSoC boards.

One important part of this project is for students to be able to browse through microcontroller documentation. To reach that objective, information and details about the PSoC microcontroller are purposely left out of this assignment, in order to force the student to find help in online and offline documentation. Teaching assistants can provide support for questions about system design, but will not help with questions that are related to implementation or coding of PSoC functionalities.

1.2 Available hardware

In this project, you will dispose of the following hardware elements.

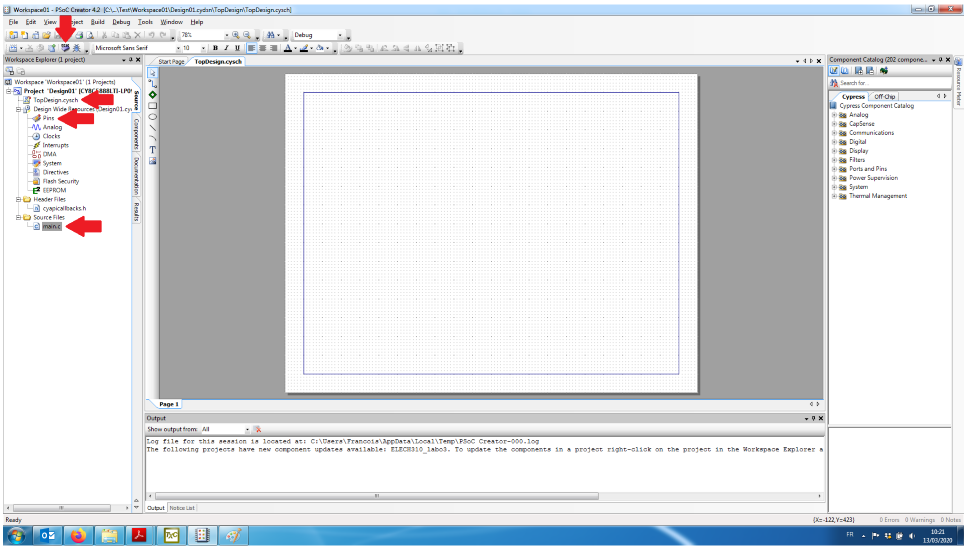
- An PSoC board (CY8CKIT-059) with a custom ULB-designed extension board (containing switches, a potentiometer, an audio jack output, LEDs, ...). The details of the extension board can be found in the schematics at the end of this document;
- two servomotors;
- a photoresistor (i.e. a light probe);
- a 2.7 kΩ resistor;
- a keyboard;
- a LCD screen (on the extension board);
- a serial-to-USB converter to allow your computer to communicate over the serial port;
- a small breadboard;
- 15 male-male cables.

The PSoC board is shown in Figure 1. It consist of the PSoC microcontroller (i.e. the integrated circuit in the center of the board) and a programmer (i.e. the integrated circuit on the left side of the board). The programmer allows to upload the program to the microcontroller directly from a host computer. Figure 1 also show two pins that can be used to provide a 5 V and a GND signal (while allowing to provide some current) directly from the host computer USB port.

2 Software requirements

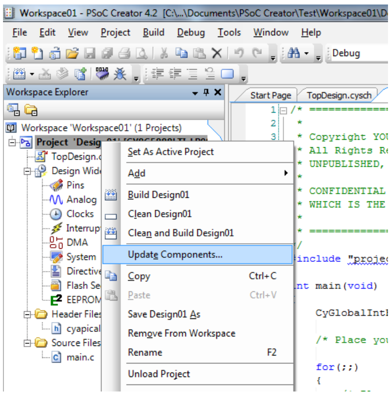
To program the PSoC microcontroller, you need to install the PSoC Creator software suite. You can download this software on the following website: `https://www.cypress.com/products/psoc-creator-integrated-design-environment-ide`. The software is quite large, so it might take a while to download. You can then install the software suite. You should also download the CY8CKIT-059 Kit setup on `https://www.cypress.com/file/416376/download` (you might need to create an account to download this file). Once downloaded, you should execute the downloaded file to install the CY8CKIT-059 Kit.

When launching the PSoC Creator IDE, it will usually ask you to register. You can ignore this by selecting “Register later”. When creating a new project, you need to specify which microcontroller you are using. In this project, we will use the “CY8CKIT-059 (PSoC 5LP)”. Once your project is created, your project will be loaded as shown in Figure 2.



```
Log file for this session is located at: C:\Users\Francois\AppData\Local\Temp\PSoC Creator-000.log
The following projects have new component updates available: ELECH310_labo3. To update the components in a project right-click on the project in the Workspace Explorer a
```

Figure 2: PSoC Creater IDE.



```
/* ============
 *
 * Copyright YOU
 * All Rights Re
 * UNPUBLISHED,
 *
 * CONFIDENTIAL
 * WHICH IS THE
 *
 * ============
*/
#include "projec

int main(void)
{
    CyGlobalIntE

    /* Place you
    for(;;)
    {
```

Figure 3: How to update the PSoC firmware.

In this project, you will mainly be using three tabs of the IDE:

- **TopDesign.cysch:** In this window, you will instantiate all the hardware elements that you will use in the microcontroller, i.e. the GPIOs, the ADC, the DAC, the PWM, the LCD screen, etc.
- **Pins:** This window will help you to associate each hardware element you instantiated to an actual output pin of the PSoC board. The numbers of each pin is written on the PSoC board itself, and you can use the document `Extension_PSoC.pdf` to see which pin of the PSoC

board was connected to which input/output of the custom-designed extension board. One of the particularities of the PSoC board is that each hardware element can be connected to any pin.

- **main.c:** In this window you will write the firmware, i.e. the actual program that will be executed on the microcontroller. The code mainly consists of a main function that contains an initialization code and an infinite for-loop. Your microcontroller will continuously execute the program that is written in the infinite for-loop until the microcontroller is turned off.

To upload your program to the PSoC board, connect the PSoC programmer to your computer with the USB cable, and click the "Program" button. This will build, compile and upload your project to the microcontroller. Useful documentation can be found on the following websites:

- Getting Started with PSoC 5LP: `https://www.cypress.com/file/41436/download`
- Tutorials for each hardware element of the PSoC: `www.cypress.com/psoc101`
- Video example on how to use the PSoC: `https://www.cypress.com/video-library/PSoC`

It is possible that the first time you try to upload a program to the PSoC board, you get an error message. This is because the firmware on the PSoC board needs to match the software version of PSoC Creator IDE. To solve this problem, right-click on the project and select "Update components" (see Figure 3. Then just follow the instructions with the default settings.

3 Methodology

In the following sections, we will introduce the different hardware elements that can be instantiated on the PSoC microcontroller, and how to control these hardware elements from the microcontroller code. You will need all of these elements to solve the assigned task. Of course, the following sections are merely an introduction, not full tutorials. Small assignments will be given to learn how to use the components, but these assignments are not per se directly related to the project. Please refer to the PSoC tutorials (`www.cypress.com/psoc101`) or the datasheet of the hardware elements (these can be found by double-clicking on a component in the “TopDesign.cysch” file, and clicking on the button “Datasheet”).

3.1 General purpose input-output (GPIO)

GPIOs are the most basic hardware elements, allowing to provide input commands to the microcontroller and receive output signals from the microcontroller. You can find tutorials of how to use GPIOs on the following website:

- `www.cypress.com/psoc101`, Lesson 1: "1. Software Output Pins"
- `www.cypress.com/psoc101`, Lesson 2: "2. Software Input Pins"

The following short assignment will help you to interface two sort of IOs : push-buttons and LEDs. The short assignment has the following specifications: the LEDs D1-D4 must switch on when button SW1 is pushed, and switch off when this button is released.

- Locate the different peripherals present on the extension board and named in the annex `Extension_PSoC.pdf` (LEDs, push-buttons, potentiometer, analog output connected to power amplifier, etc.).
- Identify which PSoC pin number connects to each LED and each switch of the extension board.
- Create a new project in PSoC creator.
- In the TopDesign.cysch file, instantiate the LEDs as digital outputs (use the "Strong Drive" mode[1]). When double-clicking on a component in the TopDesign.cysch file, you open it's customized menu, where you can select the options of the component. Similarly to the video tutorial on PSoC 101, you can add the resistors, diode and ground of the extension board as off-chip component for more clarity in your design (use the schematics in the annex «Extension PSOC» for exact component values and mapping).
- In the TopDesign.cysch file, instantiate the switches as digital inputs (use the "High Impedance Digital" mode). Again, you can add the elements of the extension board as off-chip components in your design for more clarity.
- In the projectName.cydwr file, associate each LED and each switch the appropriate PSOC pin number you identified before.
- Now go the file where you write the firmware code (i.e. main.c). You can now write the code fulfilling the specifications:
  - Locate the for(;;) loop. This endless loop contains the tasks that must run continuously, while the code placed before and after this loop is only executed once and is mainly used to configure the peripherals and define the variables.
  - Write how the LEDs and switches should behave. When double-clicking on a component in the TopDesign.cysch file, you open it's customized menu. There is a button to rapidly access the datasheet of the component. Use the "Application Programming Interface" section to determine how to use the component from the firmware (i.e. the main.c-file).
  - You can use the function `CyDelay()` to put the microcontroller to sleep for a certain amount of milliseconds between two iterations of the for loop.
  - Build, compile and load it on the μC board, then check its behavior.

It is important to note that, in general, microcontroller are not designed to deliver high currents. If we were using high-brightness LEDs, we would need to use *buffer* circuits (such as the `74ACT244`) to provide larger currents.

[1] this mode allows to output some current from the microcontroller, which is required to light the LEDs

3.2 LCD screen

The LCD screen is a particular output that allows you to write characters on the 16-characters LCD screen, located on the extension board. By examining the annex `Extension_PSoC.pdf`, you will notice that the LCD requires 7 output ports from the microcontroller. On the left side of the LCD screen, there is a potentiometer that allows you to control the contrast of the LCD screen. The 8 first characters of the LCD screen are defined to be row 0, the 8 last characters of the LCD screen are defined to be row 1. The exact mapping of the rows and columns of the LCD screen are shown in Figure 4. To use the LCD screen, follow the following steps:

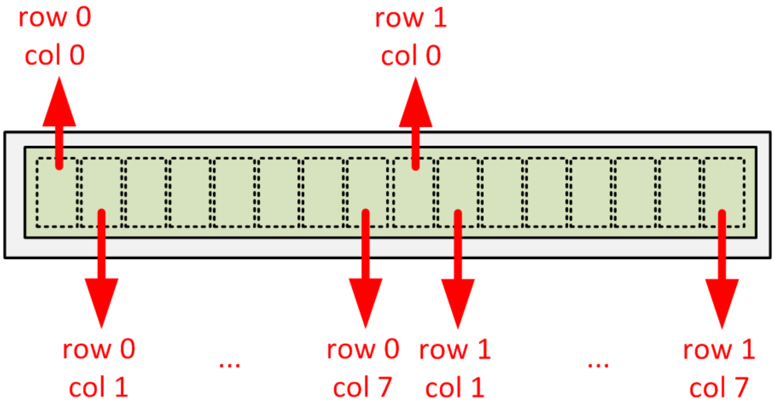


Figure 4: Row and column mapping of the LCD screen.

- Instantiate a “Character LCD” in your TopDesign.cysch file. Connect the LCD inputs to the appropriate microcontroller outputs using the “Extension PSOC” file.
- By using the API described in the datasheet of the LCD, write something on the LCD screen. The following (partial) list of functions will be particularly useful:
  - `LCD_Char_ClearDisplay()`: to clear the LCD display;
  - `LCD_Char_Position()`: to control the cursor position of the LCD display;
  - `LCD_Char_PrintString()`: to print a string of characters;
  - `LCD_Char_PrintNumber()`: to print the decimal value of a 16-bit value;
  - `LCD_Char_PrintInt8()`: to print a two-ASCII-character hex representation of the 8-bit value;
  - `LCD_Char_PrintInt16()`: to print a four-ASCII-character hex representation of the 16-bit value;

  Depending on which function you use to print on the LCD screen, it might be required to do a type conversion first.
- Try printing some characters and some values on the LCD screen.

3.3 Keyboard

A small 4 × 3 keyboard is provided, that can be connected directly to the bottom of the extension board[2]. Functions that allow to initialize the keyboard and to receive the pressed key are provided (i.e. the `keypad.c` and `keypad.h` files). The following short assignment will help you in using the keyboard.

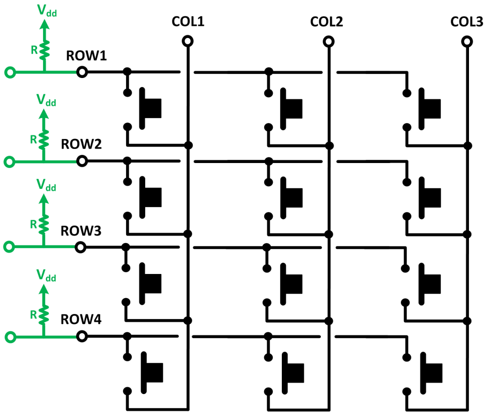


Figure 5: Electronic circuit of a 4 × 3 keyboard. The green part represents the resistive pull-ups inputs of the PSoC board.

- Figure 5 shows the electronic circuit of a matrix keyboard. Moreover, functions allowing you to read the pressed key are provided (`keypad.c` and `keypad.h` files). For these functions to work, the inputs ROW0, ROW1, ROW2 and ROW3 must be configured as "Resistive pull-up", effectively adding the green components in Figure 5 to the circuit. When the key is effectively pressed, the function `keypadScan()` returns the corresponding character ('0' to '9', '*' or '#'). Otherwise, it returns the value 'z'. Using the electronic circuit of the matrix keyboard and the code provided, understand how this matrix keyboard works.
- In your TopDesign.cysch file, instantiate COL1, COL2, COL3 as digital output with "Open Drain, Drives low". Instantiate ROW0, ROW1, ROW2 and ROW3 as Digital inputs with "Resistive pull-up" (make sure the "HW connection" box is unticked).
- Add the `keypad.h` file and the `keypad.c` files to the folder where your `main.c` is located (i.e. Documents/PSoC Creator/MyProject/MyProject.cysdn/). In your PSoC workspace, you should add the `keypad.c` file to the "Source Files" folder, and the `keypad.h` file to the "Header Hiles" folder. Finally, add the following line at the beginning of your firmware code (i.e. the main.c file):

```
#include "keypad.h"
```

- Now write a code that prints successive pressed keys on the LCD screen. Your code should scan the keyboard every 100 ms. Note that if the keyboard is pressed only once, the value should be printed only once on the LCD screen.

[2] there are 8 pins to the bottom of the extension board to allow for 4 × 4 keyboards. Please use the seven rightmost pins for a 4 × 3 keyboard.

3.4 Analog-to-digital converter (ADC)

Microprocessors are present in practically every industrial processes (temperature control, speed display, alarm systems) as well as in our common life (alarm-radio, audio systems, etc.). All those processes have in common the interaction with analog variables. Processors work with digital variables thus, it is necessary to convert those physical analog quantities into digital values and inversely.

The transformation of physical signal to digital is done in two steps :

1. Conversion of the physical signal to measure (temperature, speed, ...) into voltage. It is the role of the sensor (thermocoupe, accelerometer, etc.).
2. Conversion of this voltage into binary value. It is the role of the analog-to-digital converter (ADC).

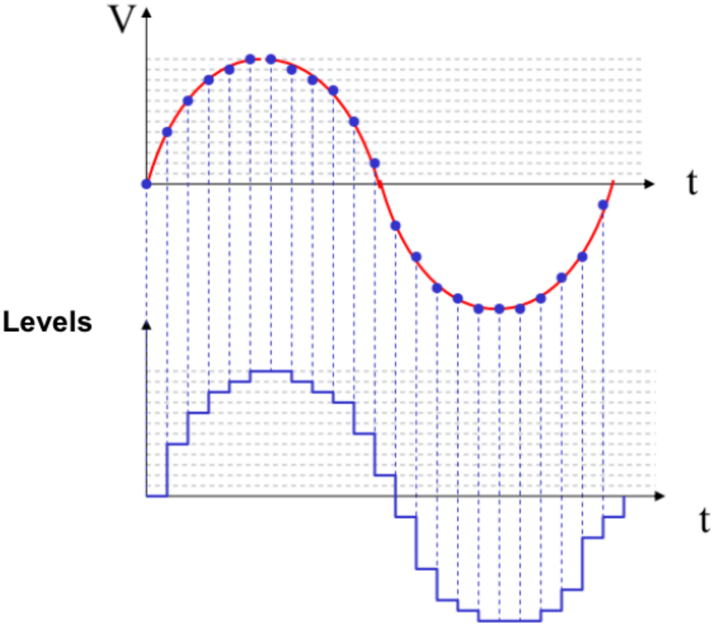


Figure 6: Double quantization

The analog-to-digital converter turns an analog voltage (changing continuously) into a digital number coded on a fixed value of bits (8 to 20 bits for the PSoC). To do so, the converter does a double quantization (see figure 6): 1) a time quantization (also called sampling), and 2) a level quantization. Note that it is impossible to perfectly represent an analog signal into a finite number of bits. The performances of an ADC depend mostly on two parameters :

- The sampling frequency.
- The resolution, in other words the number N of bits coding the converted value. The resolution can be electrically defined as the smallest detectable change of voltage. If the converter has an operating range from 0 V to 5 V and convert the quantity on 10 bits, the resolution is $\frac{5V}{2^{10}} = 4,9mV(i.e.$ input range. Finally, each binary number coded on N bits will correspond to an input voltage range.

3.4.1 Potentiometer

In this short assignment, we will use the ADC to measure the voltage on a potentiometer connected to input `P0[7]` (see `Extension_PSOC.pdf`).

- Instantiate an (analog) input pin for the signal coming from the potentiometer in the TopDesign.cysch file.
- Instantiate a Delta-Sigma ADC in the TopDesign.cysch file. The ADC should sample at 10 kHz with a resolution of 16 bits. Since the signal we are measuring has the same ground as the PSoC board, the ADC should be set to Single-ended mode rather than differential. Make sure to select the appropriate mode to perform a continuous ADC conversion.

- Using the ADC datasheet Application Programming Interface section, determine which function you should call from the firmware (i.e. the `main.c` file) to start the ADC (this should be done in the initialization code, not in the infinite loop).
- Next, determine which function you should call to start an ADC conversion.
- Next, determine which function you should call to determine whether the ADC has finished it's current conversion.
- Next, determine which function you should call to read the ADC value. Should you use the 16- or 32-bit version of the function?
- Finally write the firmware code (i.e. the `main.c` file code) to launch an ADC conversion, check whether the ADC is finished converting, read the ADC value and write it on the LCD screen. Don't forget to start your ADC before the infinite loop!

3.4.2 Photoresistor

In this project you are required to use a photoresistor to measure the luminous intensity. A photoresistor is a component whose resistance varies with luminous intensity. It has a value around 10 kΩ in the dark, and a value below 3 kΩ when the light is turned on. To use this photoresistor, you should use the electronic circuit shown in Figure 7. The photoresistor is mounted on a voltage divider. The (variable) photoresistor will determine the amount of current that flows through the two resistors, hence the voltage drop over resistor R_1. The value of R_1 is 2.7 kΩ or 3.3 kΩ, such that the voltage drop over R_1 can be measured with the PSoC ADC for all possible values of the photoresistor. If you use a single-ended ADC, don't forget to connect the ground of your circuit to the ground of your PSoC device.

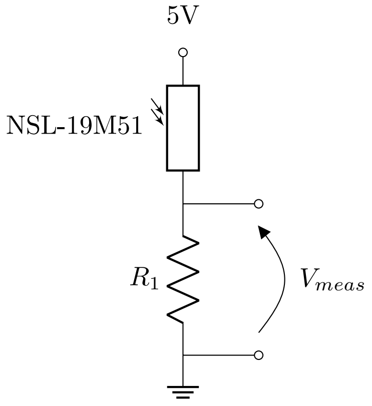


Figure 7: Luminous intensity measurement with a NSL-19M51 photoresistor.

In this short assignment, we will measure luminous intensity using a photoresistor.

- Use the breadboard to realize the circuit of Figure 7. The 5 V should be taken from the `POWER` pin on the PSoC device (see Figure 1). The ground can be connected to any ground pin of the PSoC device or extension board.
- Connect the output voltage of the circuit to an (analog) input port of the PSoC device (use the GPIO inputs on the right of the extension board).
- Use a Delta-Sigma ADC to measure the analog voltage.
- Turn the LEDs on when the light is off, turn the LEDs off when the light is on.

3.5 Timers

A timer is a hardware element that is connected directly to the microcontroller oscillator, and is able to provide precise time management for the microcontroller. The `CyDelay()` function allows to put the microcontroller to sleep for precise time durations, but the programmer cannot know the execution time of other instructions. Hence, precise time management is impossible to do without timers.

In the following short assignment, you are asked to make the LEDs blink at a given frequency. To do so, you will need a timer. This type of peripheral has a 8/16/24/32-bits register that is decremented at each clock cycle. When the value of this register reaches 0, the counter goes back to the specified period value and starts decrementing again. At the same time, a specific bit named `TC` (in the `Timer_Status` register) goes to 1 to warn of overflow (cf. datasheet of timer). As a first step, you will write a program allowing to make the LED blink at a frequency of 1 kHz.

- Instantiate a 16-bit Timer in the TopDesign.cysch file (use the “Fixed Function” implementation). Set the period to cause an overflow at a frequency of 1 kHz. What is the largest period of the timer? As a reminder, the highest clock rate available in a PSoC system is a BUS_CLK at a rate of 24 MHz.
- Add the launching of the timer in your program (use the Datasheet Application Programming Interface to find the appropriate function).
- In the main loop of your program :
  - Check by polling if the timer has overflowed (ie. If the bit `TC` has risen to ’1’). You can do so by using the following instruction:

    ```
    overFlow = 0x80 & Timer_ReadStatusRegister();
    ```

    Explain how this instruction works? How is this line of code able to extract the bit `TC` from the `Timer_Status` register? The datasheet mentions that the bit `TC` is “sticky”. This means that you do not need to reset the bit `TC` to zero manually, it is reset automatically to zero after reading it.
  - Write a function that toggles LED4 at the overflow frequency.
- Since the human eye cannot distinguish a LED oscillating at a frequency of 1 kHz, you cannot explicitly verify if your code works well without an oscilloscope. However, you should notice that the LED does not shine as brightly as in the previous assignments.
- Modify your code to obtain a period to 1 s for the blinking of the LEDs. Since a 1 s period cannot be achieved directly with a timer (see maximum period above), we rely on the code to count overflows of the timer until a 1 s period is reached.

3.6 Digital-to-analog converter (DAC)

The principle of the DAC is the opposite of the ADC : the transformation of a number coded on N bits to a voltage in its output range. The voltage obtained is continuous in time (the output of the converter is constant until a new conversion is done), but is always quantized because only 2^N different voltage values are achievable (each binary code correspond to one voltage).
In the following short assignment, you are asked to realize a synthesizer capable of generating an arbitrary sound wave and convert it to sound through a speaker. In our case, we will simply generate a sine wave at adjustable frequency : the processor will send samples corresponding to the wanted waveform to a DAC and once rebuild, the signal will be send to the amplifier of the extension board (element U3 on the extension board). You can listen to the generated sound by connecting speakers to the audio jack.

- During the configuration of your peripherals (before the infinite loop), fill a vector (of length 100) containing the sound wave, defined as global variable so that it contains one full period of a sine wave defined between -1 and +1. To define the vector as a global variable, it needs to be declared outside of the main function. To do so, you must :
  - include the file math.h in the header ;
  - add the math library to the linker, right click on the project > Build settings... > ARM GCC > Linker > General > Additional Libraries and add ”m” ;
  - call function `sin()`, that takes as argument a floating number representing the angle (in rad) and that also returns a floating number.
- Program a timer so that a sample of the sine wave is processed every 50 μs
- Instantiate an (analog) output pin of the DAC in the topDesign.cysch file, and connect it to the `DAC_OUT` pin of the extension board (see `Extension_PSOC.pdf` file).
- Instantiate a VDAC in the TopDesign.cysch file. The range is from 0 to 1.020 V and a slow speed is sufficient. Make sure you select “CPU” as the data source, since we will provide the data from the firmware (i.e. the `main.c` file). The strobe should be set to “Register Write”, which means the DAC will provide a new analog value every time a new value is written in it’s register.
- In your firmware code (i.e. the `main.c` file), poll the timer to detect timer overflows (as done in the previous lab). Every 50 μs, write the next value of the sine wave vector to the DAC by using the `DAC_SetValue()` command (that you can find in the VDAC datasheet Application Programming Interface). Note that the DAC expect to see an 8-bit signal (i.e. between 0 and 255), so you should shift (center around 128) and scale (amplitude between 0 and 128) your sine wave so that it remains within this range at all times. Don’t forget to start your DAC before the infinite loop!

3.7 Interrupts

We will now study how we can use interrupts in our code to generate super-priority events. When an interrupt is latched, the firmware code in the `main.c` is temporarily interrupted while the Interrupt Service Routine (ISR) is executed. You can follow the tutorial on the following website for an introduction to interrupts: `www.cypress.com/psoc101`, Lesson 3: “3. Interrupts”.

- Take your previous project on the Digital-to-Analog Converters.
- To write the data to the DAC from the firmware, we will now create an Interrupt. Instantiate an Interrupt in the TopDesign.cysch file, and connect it to the interrupt-output of the timer. This means that every time the timer overflows, this interrupt will be triggered.
- Following the video tutorial on interrupts, write your firmware so that your code that was previously inside your polling-loop is now in your ISR (don’t forget to remove/comment out the polling of the timer). Read the documentation about the interrupt output in the timer datasheet. After an interrupt is triggered, the interrupt output remains asserted until the status register is read. To clear the status register, use the function `Timer_ReadStatusRegister()`.

3.8 Pulse Width Modulation (PWM)

A PWM is a module that can send a square wave signal (such as shown in Figure 8) to a DC motor or to a servo motor. For controlling a DC motor, the ratio of high versus low state of the PWM wave determines the speed of the DC motor. For a servo motor, the length of the pulse determines the position of the servo motor. In order to use the PWM, check the tutorial on described in www.cypress.com/psoc101, Lesson 6: "6. Pulse Width Modulator". The principle of a PWM can be described as being a timer with two thresholds:

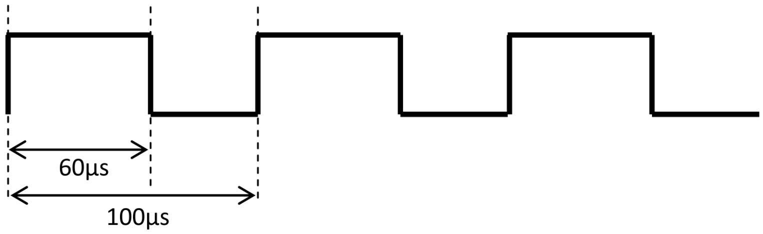


Figure 8: PWM Signal with a duty cycle of 60 %.

- When the timer starts, the PWM output pin `pwm` is at '1'.
- When the counting register of the timer reaches the first threshold, named `CMP`, the output `pwm` turns to '0'. The timer continues to increment.
- When the counting register reaches the value of the period `Period`, the timer returns zero and the output returns to '1'.

By playing with the values of `CMP` and `Period`, it is possible to generate any square wave.
The following short assignment will allow you to control a servo motor with a PWM module.

- Connect the servo to the microcontroller (see the servo datasheet on `http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf`). The orange cable should be connected to an output of J1 or J2, the black cable to the ground, and the red cable to the 5 V output pin (note that the servo consumes some current, so you cannot connect the red cable to any GPIO set to the high state).
- Instantiate a PWM in the TopDesign.cysch file. The servo datasheet indicates that you need a period of 20 ms for the PWM square wave. However, by default, the maximum period of a 16-bit PWM module is 2.731 ms. To increase this range, you need to double-click on the `CLK24MHZ` block in the TopDesign.cysch. By increasing the divider value in this block, you effectively reduce the rate of the clock driving the PWM, and you increase the possible `Period` range of the PWM. Adjust the values until you reach a `Period` of 20 ms.
- Adapt the `CMP` value for the pulse to have a high-value of 1.5 ms.
- By using the function `PWM_WriteCompare()`, change the duration of the pulse. What happens with a pulse of 1 ms or 2 ms?
- Use the potentiometer and ADC code from the ADC assignment to make the duration of the pulse vary from 1 ms to 2 ms depending on the position of the potentiometer.
- Validate that the servo is turning accordingly.

3.9 Serial port communications

The serial bus allows to send and receive data from/to the microcontroller to/From the PC. This module is named UART (Universal Asynchronous Receiver / Transmitter). Before designing this communication, you will have to set the PSoC and the PC in order to make them "speak the same language". We will start by the PSoC setup. Add an UART component to the TopDesign.cysch file. Connect the rx and the tx port of the UART component to one of the J1 or J2 GPIO ports on the PSoC extension board. Double-click on the UART component to set the UART communication parameters. The parameters of the UART connection are the following:

- Baudrate: 9600
- Data bits: 8
- No parity bit
- Only one stop bit
- No flow control

3.9.1 Sending data to the PC

To send and receive the data on the PC side, use a serial communications software (such as "Termite 3.4", that you can download on `https://www.compuphase.com/software_termite.htm`). This software allows to connect an USB-to-serial converter to the PC, and exchange data with an UART device. Make sure the UART communication parameters match that of the PSoC UART component. To determine the right COM port, connect the UART-to-USB converter to the PC and do the following :

- Wait for Windows to install the drivers
- Open the Device Manager by pressing the Windows Key + R. Type "devmgmt.msc" and press Enter.
- Expand the Ports (COM & LPT) section.
- Find the COM port associated to the USB-to-serial (or USB-to-UART) converter.

Connect the UART-to-USB converter between the PC and the PSoC (**note that the PSoC Tx port correspond to the UART-to-USB Rx port, and vice-versa !!!**). Do not forget to connect the ground of the PSoC to the ground of the UART-to-USB converter.
To send data to the PC from the PSoC device, refer to the UART datasheet. The following tasks need to be performed:

- Start the UART transceiver.
- Test the UART transmission with the `UART_PutString()` command
- Send specific data over the UART by using the `UART_PutChar()` command. To convert an (single-digit) integer to a ASCII character, you can just add the term '0' (see code snippet below). Based on how ASCII characters are coded, can you explain why this code snippet works?

```
value_char = value_int + '0';
```

3.9.2 Sending commands from the PC

We want the PSoC to trigger an interrupt each time a byte is received from the PC on the UART port. To do this,

- Check the "RX - on byte received" box in the UART component in your TopDesign.cysch file
- add an "Interrupt" component connected to the `rx_interrupt` in the TopDesign.cysch file
- In the firmware (i.e. `main.c` file), don't forget to start the interrupt related to the UART reception
- Write the code for the UART ISR, based on the sample code below. Try receiving characters from the PC and print them on the LCD.

```
CY_ISR ( isr_uart_rx_Handler ) {
    uint8_t status = 0;
    do{
        // Checks if no UART Rx errors
        status = UART_ReadRxStatus();
        if ((status & UART_RX_STS_PAR_ERROR) |
            (status & UART_RX_STS_STOP_ERROR) |
            (status & UART_RX_STS_BREAK) |
            (status & UART_RX_STS_OVERRUN) ) {
            // Parity, framing, break or overrun error
            LCD_Position(1,0);
            LCD_PrintString("UART err");
        }
        // Check that rx buffer is not empty and get rx data
        if ( (status & UART_RX_STS_FIFO_NOTEMPTY) != 0){

            rxData = UART_ReadRxData();
            // Here comes your code, ...
            // ... what do you do with rxData ?
        }
    }while ((status & UART_RX_STS_FIFO_NOTEMPTY) != 0);
}
```

4 Project specifications

For the project, you are free to choose one of the following two projects. You are also free to propose your own project (with the elements you have in the lab kit, or with other if you have some additional elements at home). In that case, please contact the teaching staff so that we can agree on the specifications of your self-proposed project.

4.1 Distress beacon specifications

The aim of this project is to build an emergency beacon that is capable of sending three types of distress signals, as shown in Figure 9:

1. a luminous signal sending morse code;
2. an audio signal sending morse code;
3. a visual flag type signal sending semaphore code.

Morse code is a succession of short and long luminous or audio signals, which can describe the letters of the alphabet. Semaphore code uses two flags to describe the different letters of the alphabet: `https://gregheo.com/images/blog/semaphore-alphabet.jpg` (note that your servos can only rotate 180°, so not all letters can be done. You are free to change the combination of flags for those letters). Different inputs will be used to allow different functionalities of the distress beacon.

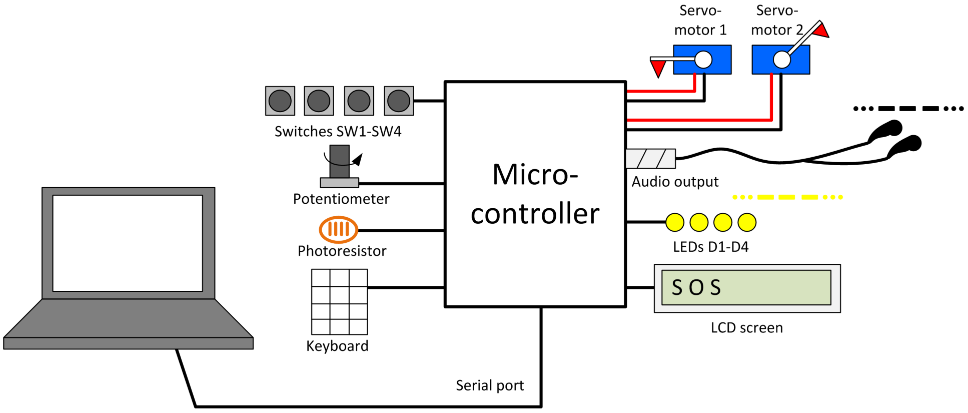


Figure 9: Schematic representation of the distress beacon inputs and outputs.

The inputs should be configured as follows.

- **SW1:** When pressing switch SW1, the LEDs should light up and the audio output should emit a sound as long as the switch is pressed. When the switch is unpressed, the LEDs should go out and the audio output should stop emitting a sound.
- **SW2:** When pressing switch SW2, the LEDs and audio output should emit a short morse signal. The short morse signal should be 250 ms long.
- **SW3:** When pressing switch SW3, the LEDs and audio output should emit a long morse signal. The long morse signal should be 750 ms long.
- **SW4 and Pot:** When turning the potentiometer (Pot), the flag should be going up according to the turning of the potentiometer. When pressing SW4, we change the flag that is controlled with the potentiometer.
- **Photoresistor:** The photoresistor can detect whether there is ambient light or not. When there is light, only the audio output and semaphore flags should be active. When there is no light, only the audio output and the LEDs should be active.
- **Keyboard:** We can use the keyboard to send a series of pre-recorded message. For example, pressing “1” and “*” should start sending “SOS” in morse and semaphore code. Pressing “2” and “*” should send “BEAMS” in morse and semaphore code. Other messages are for you to chose.
- **Serial port:** The user computer can send ASCII characters through the serial port, which are then sent in morse and semaphore code by the distress beacon. The microcontroller also sends two types of characters to the user computer over the serial ports: “.” and “-” (which represents short and long morse signals that are sent by the distress beacon).

- **LCD screen:** The LCD prints the ASCII characters that are sent by the distress beacon (this feature can only be used when using the keyboard or the serial port). You can also chose to print additional information on the LCD screen.

4.2 Never-lose specifications

The aim of this project is to build a system that can play Google Chrome's *dino* game autonomously (without ever losing of course). You can play it by opening a Chrome browser and typing the following URL: `chrome://dino`. When you press the spacebar, the game will launch. Your *Never-lose* system should:

1. use the photoresistors to detect incoming obstacles on the screen;
2. use the servomotors to press the spacebar and the "down" arrow.

You can attach the photoresistors to your screen with some duck tape, the servomotors should be taped to the keyboard or held down with some weights to ensure that they don't move when pressing the computer's keys.

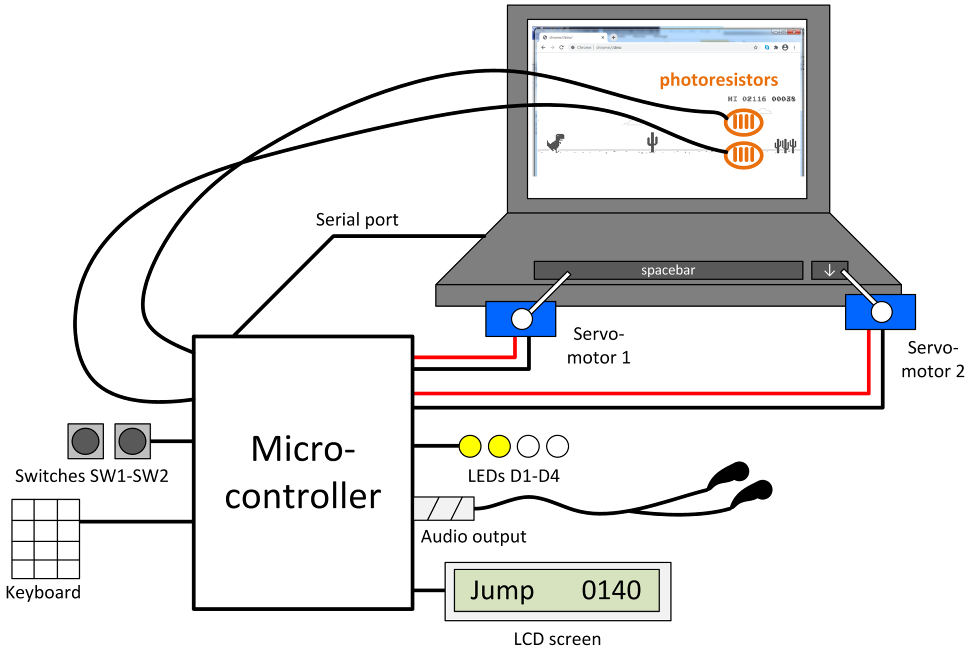


Figure 10: Schematic representation of the never-lose system.

The inputs and outputs should be configured as follows.

- **SW1:** When pressing SW1, the servo controlling the spacebar should press the spacebar.
- **SW2:** When pressing SW2, the servo controlling the down-arrow key should press the down-arrow key.
- **SW3 and score counting:** When pressing SW3, the score should be reset to zero. The score should start increasing the first time the spacebar is pressed after resetting the score to zero (in the dino game, the score increases with 10 points per second). The score should be printed on the second half of the LCD.
- **LED1-LED2:** These LEDs should light up when the dinosaur is jumping.
- **LED3-LED4:** These LEDs should light up when the dinosaur is ducking.
- **Photoresistor:** The photoresistors should be used to detect incoming obstacles (one photoresistor for the ground obstacles, one for the flying obstacles).
- **Keyboard:** You can pick two keys of the microcontroller keyboard that should have the same effects as SW1 and SW2.
- **Serial port:** The user computer can send the "jump" and "duck" command through the serial port (for example for launching the game), which should result in the correct servo pressing the correct key. The microcontroller should also send the following messages to the user computer over the serial ports: "Jump" and "Duck" when the dinosaur is jumping/ducking.

- **LCD screen:** The LCD screen should print “Jump” when the dinosaur is jumping, “Duck” when the dinosaur is ducking. The second part of the LCD should print the score.
- **Audio out:** The audio output should emit one sound when the dinosaur is jumping, another sound when the dinosaur is ducking.

As a bonus, the following elements can be implemented:

- a system to detect when the day changes into night on the game;
- ...

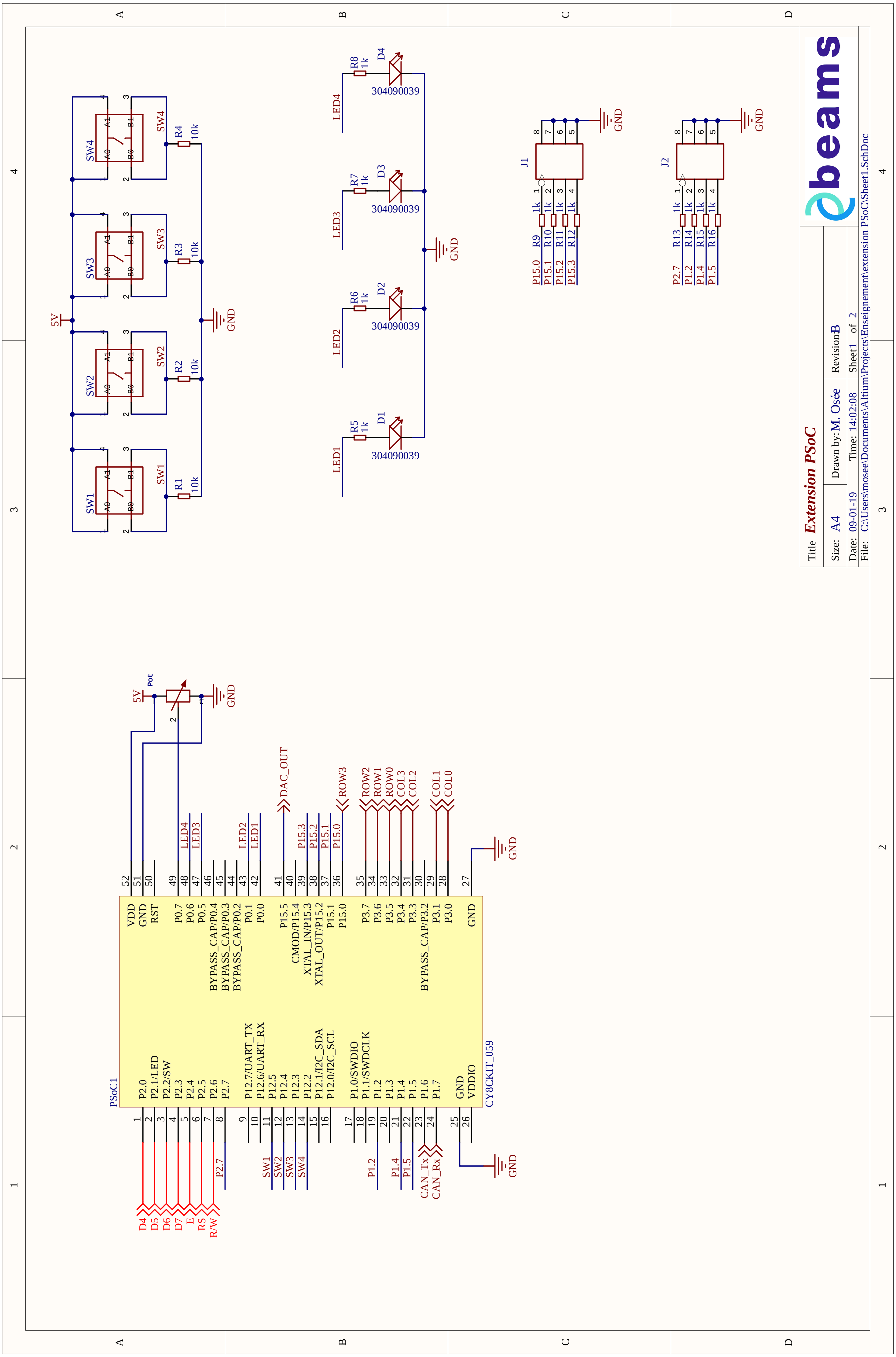
4.3 Expected deliverables

At the end of the project, we will expect the following deliverables:

- A diagram showing the state machine (including all transitions between states based on the inputs/outputs) that you used for your project;
- A fully-functional code that implements the project specifications;
- A demo (or a video that you can take with your smartphone) of the prototype in action.

A Extension PSoC

The following pages contain the detailed schematics of the custom-made extension board for the PSoC device.

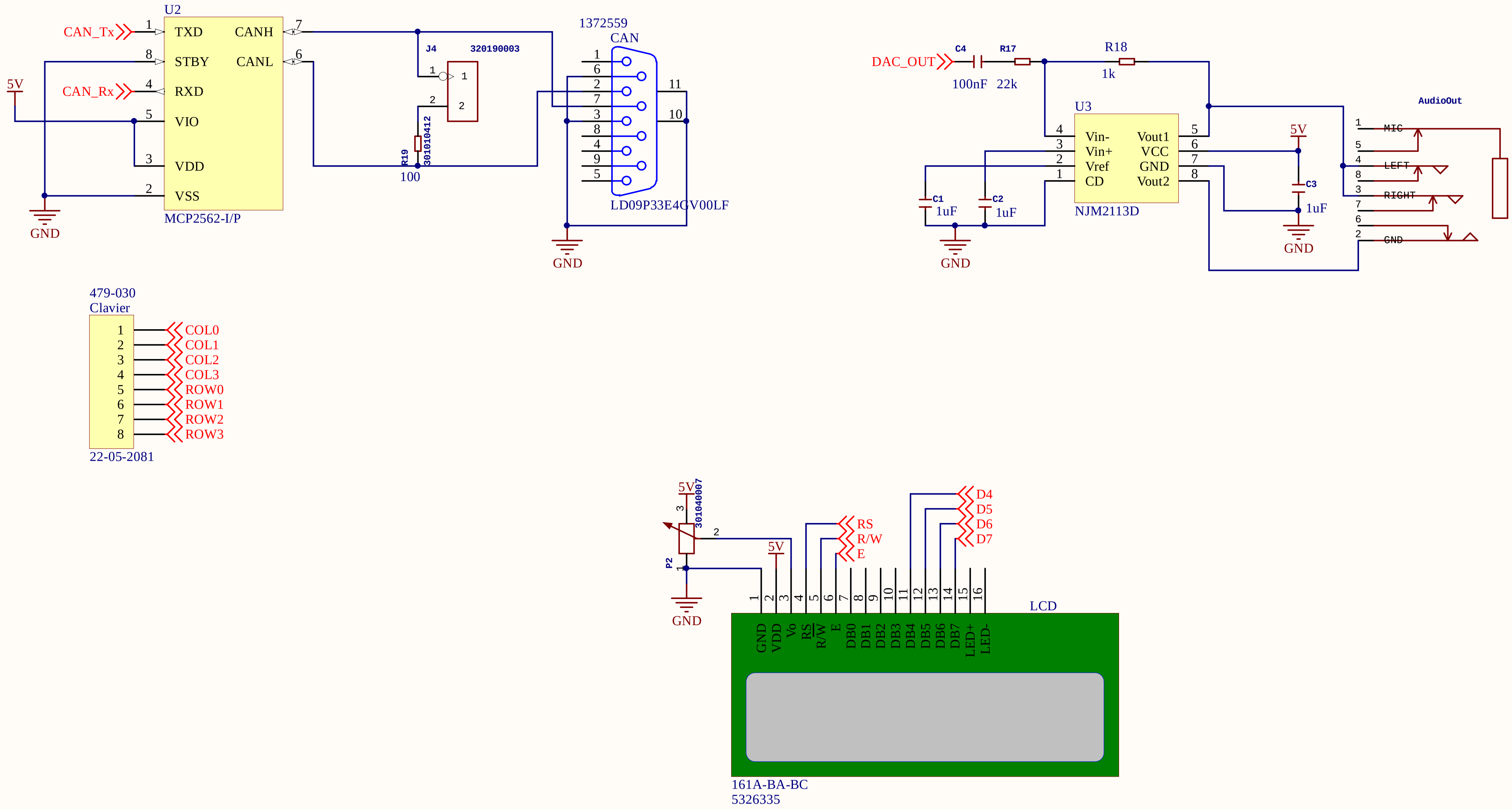


Title **Extension PSoC**

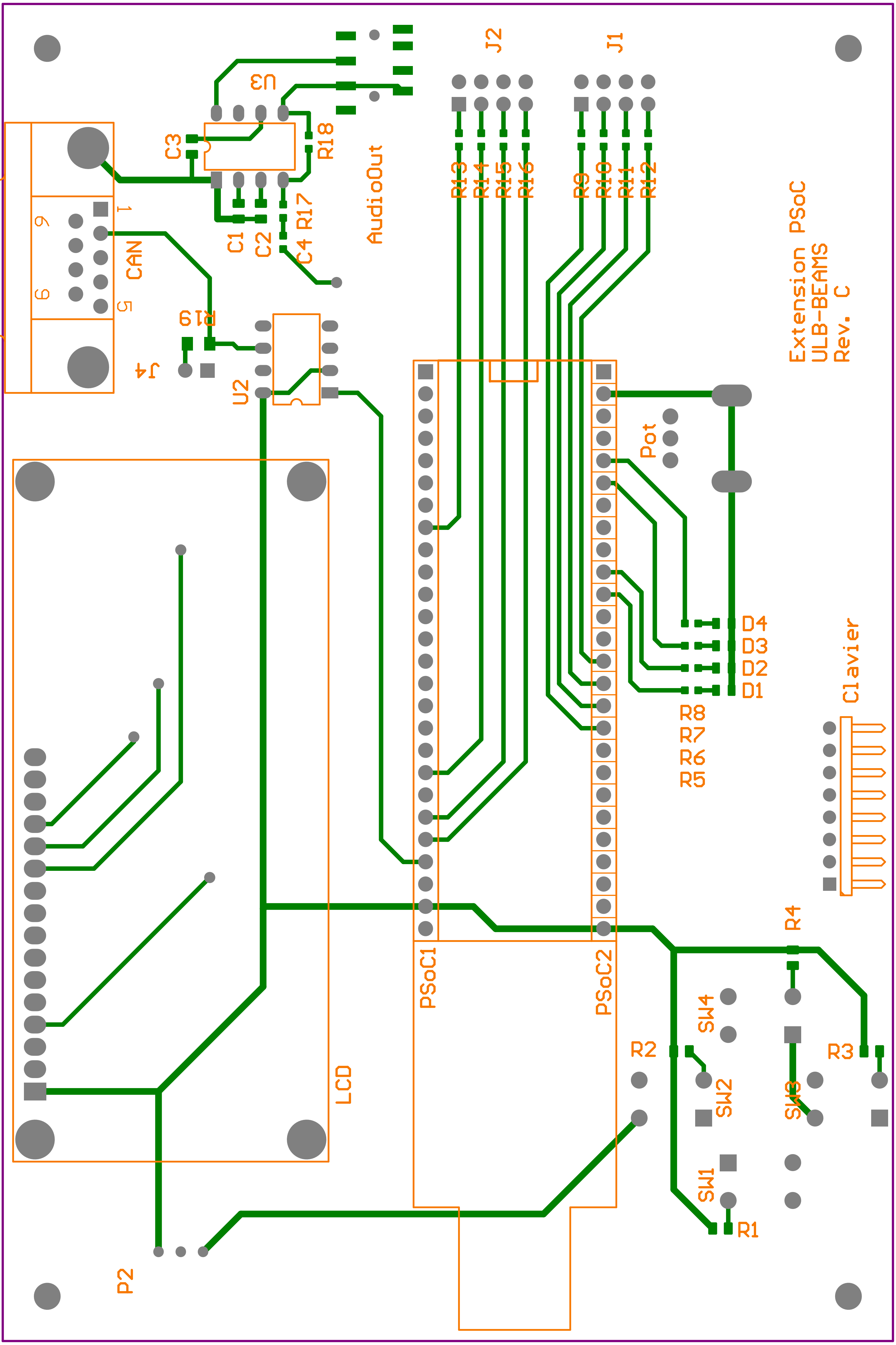
Size: A4 Drawn by: M. Osée Revision: B

Date: 09-01-19 Time: 14:02:08 Sheet1 of 2

File: C:\Users\mosee\Documents\Altium\Projects\Enseignement\extension PSoC\Sheet1.SchDoc



Title **Extension PSoC**		
Size: A4	Drawn by: M. Osée	Revision: B
Date: 09-01-19	Time: 14:02:08	Sheet2 of 2
File: C:\Users\mosee\Documents\Altium\Projects\Enseignement\extension PSoC\Sheet2.SchDoc		


Extension PSoC
ULB-BEAMS
Rev. C
J1
J2
U3
C3
R18
AudioOut
R13
R14
R15
R16
R9
R10
R11
R12
C1
C2
C4
R17
CAN
1
6
9
5
R19
J4
U2
Pot
D4
D3
D2
D1
R8
R7
R6
R5
Clavier
R4
PSoC1
PSoC2
SW4
R2
SW2
SW3
R3
SW1
R1
LCD
P2