

# Lab 4: PWM and USBUART

---

## Objective

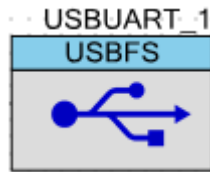
Demonstrate knowledge of using PWM and USBUART.

## Background

### B1: USBUART

One of the simplest forms of two-way communication between devices is called RS-232. It is a serial (one bit at a time) communication protocol. Data is transmitted at a bit rate that is agreed upon before starting. A device called a UART (Universal Asynchronous Receiver/Transmitter) is used for this type of serial communication.

The PSoC 5LP Development Kit has a built-in RS-232 port and there is a UART component in PSoC Creator. However, while RS-232 was a standard in PCs in the past, most modern computers no longer have an RS-232 port built in. This could be worked around by using a USB to RS-232 connector, but the PSoC can also act as a USB device through the USBFS component.



The USBFS component is a very complicated subject, as it can act as many devices, including audio and MIDI. However, PSoC Creator has an entry in its component catalog that is called USBUART, which is a USBFS component that's pre-configured to act as a virtual UART via a device class called CDC. The only thing to configure in this component is the name, as everything else about it is already configured to work, aside from the system clocks.

The USBFS component requires the system clocks to be modified so USB\_CLK is enabled and running at 48 MHz and ILO needs to run at 100 kHz. Additionally, USB\_CLK needs to have an accuracy of +/- 0.25% or better, which is not provided by the default internal oscillator settings. Configuring the clocks will be covered in the procedure.

The USBFS component generates drivers to be used by Windows to interface with the PSoC. The driver is found under Generated\_Source/PSoC5 in the project directory with the file extension .inf. For a USBUART configuration with the name USBUART\_1, it is called USBUART\_1\_cdc.inf.

Here are the most important functions in the programming interface. "USBUART" is replaced with the name given to the component in the schematic view.

void USBUART_Start (uint8 device, uint8 mode)	Starts the USB device. device will usually be 0 and mode indicates the voltage that should be used to power the PSoC. Use the mode USBFS_3V_OPERATION for 3.3v and USBFS_5V_OPERATION for 5v.
---	---

uint8 USBUART_GetConfiguration()	When this returns non-zero, the device is ready
void USBUART_CDC_Init()	Initializes the CDC interface (which is used for UART). Only call this after calling Start and GetConfiguration returns non-zero.
uint8 USBUART_CDCIsReady()	Returns a non-zero value when the device is ready to send more data. <b>Always check before sending new data!</b>
void USBUART_PutString (char8* string)	Sends the given null-terminated string to the host. If the string is longer than 64 characters, it will send the string in pieces until the end of the string is put into the output buffer. Therefore, this function will block when given a long string.
void USBUART_PutData (uint8* pData, uint16 length)	Sends the specified number of bytes of data to the host. The maximum length is 64; specifying any more will result in lost data.
void USBUART_PutChar (char8 txDataByte)	Sends the specified character to the host. PutString or PutData is more efficient for large data.
void USBUART_PutCRLF()	Sends a carriage return and line feed character, or “\r\n”
uint8 USBUART_DataIsReady()	Returns a non-zero value if a packet has been received from the host, whether it contains data or not. Even if the packet has zero data, it still needs to be read by one of the functions below to allow another packet to be read. <b>Always check before reading data!</b>
uint16 USBUART_GetCount()	Gets the number of bytes received from the host. Maximum = 64
uint16 USBUART_GetData (uint8* pData, uint16 length)	Puts received bytes into the specified buffer, with length specifying the maximum number to transfer. The return value specifies how many bytes were transferred. The maximum number of bytes received is 64. If more bytes were received than the length specified, <b>the remaining bytes will be lost</b> . Best used in tandem with GetCount.
uint16 USBUART_GetAll (uint8* pData)	Puts all bytes received into the specified buffer. The return value specifies how many bytes were received. The maximum number of bytes received is 64, so the buffer should be at least that long.
uint8 USBUART_GetChar()	Returns the first byte received. <b>Additional bytes in the received buffer will be lost.</b>

*You may now start Procedure Part A. [Return here for background for Part B.](#)*

## B2: PWM

Pulse-width modulation (PWM) is often used as a simple control signal for power or lighting. Similar to a clock signal, a PWM signal consists of a period and during the period the signal is high for a percentage of the period's duration (the duty cycle).

In the implementation on the PSoC, there is a period counter that starts at an initial period value and counts down to 0 before restarting at the initial period value. The rate at which the counter counts down is determined by the input clock, where every rising edge of the clock causes the counter to decrement. After every decrement, the counter is compared with a specified value to determine whether the output signal should be high or low. Therefore, the period is specified by the clock frequency and the initial period value and the duty cycle is specified by the comparison value and comparison type. See Figure 1 for an example PWM signal with these parameters. In this figure, the period is 255 and the comparison is set to greater than 233. The length of the period depends on the clock.

The PWM component on the PSoC shares similarities with the Timer component, as it includes a counter, a period, and a capture capability. It also has both an FF and UDB implementation, where the FF implementation uses the same hardware as the FF Timer. However, PWM has some capabilities that the Timer does not, and vice versa. The main feature that PWM has over Timer is the comparison capability, which is useful for control signaling. However, a Timer is better for simpler applications, such as timing events. The simpler Timer component can also have up to 32 bits in the UDB implementation, while PWM can only have up to 16 bits.

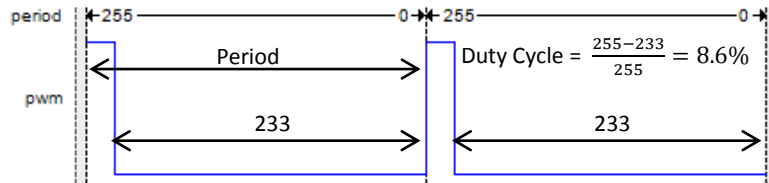
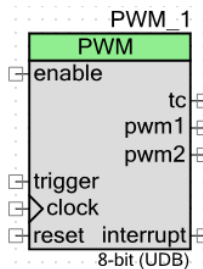


Figure 1: PWM signal with a period of 255, CMP value of 233, and CMP type of greater

The PWM component has a lot of features available that are not needed in many applications. See the datasheet for PWM for additional features, including kill, capture, and dead band.



Here are the inputs and outputs for the component:

- Inputs
  - Clock: must be connected to a clock. The period counter decrements once for each rising edge of the clock, and thus the clock and the initial value of the period counter determine the length of the period.
  - Reset: when this input is high, the period counter is reset and operation resumes when the input goes low. While this input is high, the PWM outputs are low in the UDB implementation and the outputs are high in the FF implementation.
  - Enable (UDB only): when this input is low, the period counter stops counting and the trigger input is ignored. Hidden when the Enable Mode is set to Software Only (default)
  - Trigger (UDB only): this input will trigger the component to run, depending on the run mode and trigger mode. Hidden when the trigger mode is none (default).
- Outputs
  - tc: when the period counter is 0, this output is high.
  - Interrupt: this output is high when any of the enabled interrupt sources are triggered. See the configuration options below for a trigger list. Read the status register to clear. This output may be hidden by choosing "None" under Interrupts in the configuration.
  - pwm/pwm1: The first or only PWM output. Several configuration options affect how this output behaves and whether or not there is only one PWM output.
  - pwm2: The second PWM output.

Some configuration options can be changed without rebuilding the hardware, as the software sets the initial values. These values may also be changed on the fly through the software API. These options are indicated by the word software in parentheses.

PWM components can be configured as follows (Figure 2):

- **Configure tab**
  - The top of this section depicts what the PWM signal will look like as the period counts down with the current mode, period, and CMP configurations. See Figure 1 or FIGBLAH.
  - Implementation: choose between FF and UDB
  - Resolution: 8-bit or 16-bit. Sets the resolution of the period and CMP values.
  - PWM Mode (UDB only)
    - One output: there is only one PWM output. This is the only mode in FF.
    - Two output: there are two outputs, where each has its own CMP values and CMP types. Both share the same period counter.
  - Period (Software): the starting value of the period counter. The length of the period based on the period count and current clock input in time is also displayed.
  - CMP Value 1/2 (Software): the comparison values used for determining each PWM output. Each value is compared with the current value of the period counter.
  - CMP Type 1/2 (Software): determines how the CMP value is compared with the period counter. For all of these, the comparison order is period ? CMP value, where ? is the comparison operator.
    - Less – PWM output is high if the period counter is less than the CMP value.
    - Less or Equal – High if period ctr <= CMP val
    - Greater – High if period ctr > CMP val
    - Greater or Equal – High if period ctr >= CMP val
    - Equal – High if period ctr == CMP val
    - Firmware control – generates functions to allow the software API to change the comparison type. If anything else is chosen, the comparison type is set at build time and the software API for this is removed.
- **Advanced tab (all options except Interrupts are UDB only; FF indicated for what FF uses)**
  - Enable mode: allows either software or hardware or both to control when the component is enabled (FF: software only)
  - Run mode
    - Continuous (FF): will run forever after an initial trigger
    - One Shot with Single Trigger: will run once on a trigger event
    - One Shot with Multi Trigger: will run once on a trigger event. After stopping, it will run again after another trigger event.
  - Trigger mode
    - None (FF): no trigger used; the trigger is always treated as true
    - Rising/falling/either edge: the PWM component will not start until the trigger input rises and/or falls. The software API cannot directly cause a trigger.
  - Interrupts: all disabled by default, but the software may set any of these as long as none isn't selected.
    - None: no interrupts are set. The interrupt output is hidden
    - Interrupt On Terminal Count Event: interrupt when the period counter is zero

- Interrupt On Compare 1 Event: interrupt when the pwm/pwm1 comparison value is high
- Interrupt on Compare 2 Event (UDB and proper PWM mode only): interrupt when pwm2 comparison value is high

Here are the most important functions in the programming interface. “PWM” is replaced with the name given to the component in the schematic view.

void PWM_Start()	Initializes the PWM device with default values.
uint8 PWM_ReadStatusRegister()	Reads the status register, clearing interrupts. See the datasheet for what the bits indicate.
void PWM_SetCompareMode[1,2] (enum comparemode)	Sets the comparison mode when the comparison mode type “Firmware control” is used. See the datasheet for the enum names. The name of this function has no number if there’s only one output. If there are two outputs, it ends in 1 and 2 for outputs 1 and 2.
uint8/16 PWM_ReadCounter()	Reads the current counter value (UDB only).
void PWM_WritePeriod (uint8/16 period)	Writes the period value that is reloaded after the period counter becomes 0.
void PWM_WriteCompare[1,2] (uint8/16 compare)	Writes the compare value for the given output. The name of this function has no number if there’s only one output. If there are two outputs, it ends in 1 and 2 for outputs 1 and 2.

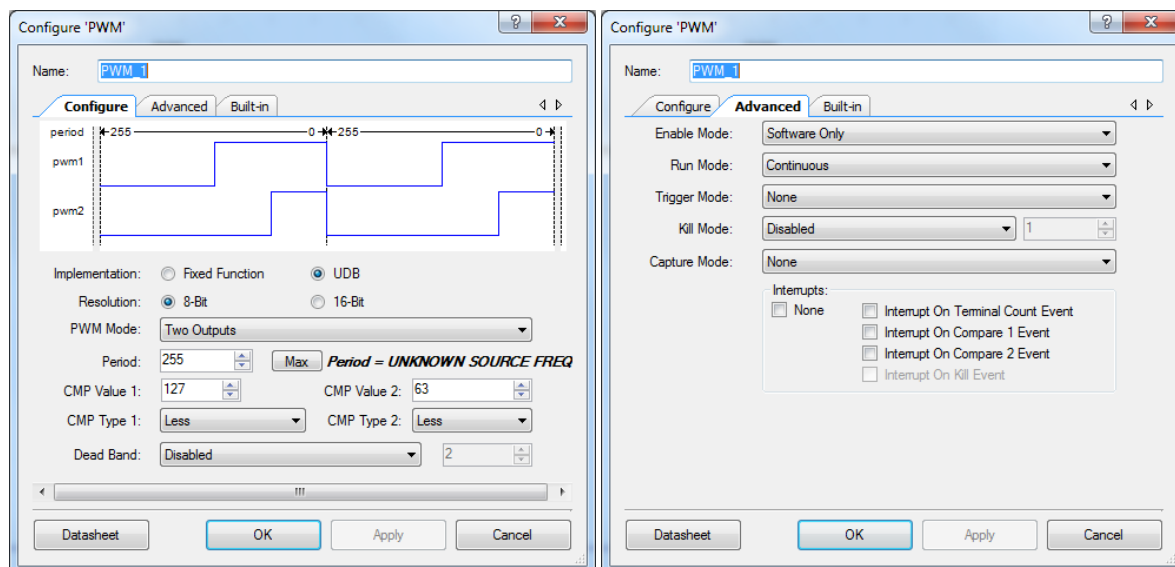


Figure 2: Configuration window for PWM

## Procedure Part A



This lab will require use of serial terminal software. This lab will use PuTTY, but you may use any software you wish. To get PuTTY, visit here: <http://www.putty.org/>  
 Note: either download putty.exe or the Windows installer. Ensure your version is at least 0.61, as 0.60 (and maybe earlier) had a bug that caused serial connections to hang.

## Outcome

In this lab, the PSoC will be programmed in two parts. This part will set up a USBUART component to be used in future parts.

### Part 1: Designing the System

1. Create a new project in your workspace named Lab4 and open its TopDesign.cysch.
2. Add a Character LCD and name it Display.
3. Add a Communications>USBUART (CDC Interface) to the schematic and name it USBUART. Other than the name, do not change anything else in the configuration.
4. Open Lab4.cydwr and configure Display to use P2[6:0]. The Dm and Dp USBUART pins will be automatically configured after application generation to P15[7] and P15[6] because those are the only pins that can be used for USB on this device.
5. Because USB is being used, the system clocks need to be changed. Go to the Clocks tab in the DWR file. Select one of the system clocks and click Edit Clock...
6. The USB clock needs to run at a fairly accurate 48 MHz. There are two primary ways to do this:
  - a. Change IMO to be 24 MHz, disable the PLL, change Master Clock to use IMO, and enable USB and have it use IMOx2. This option is shown in Figure 4.
  - b. Enable XTAL (already configured as the 24 MHz crystal on the development kit), change IMO to use XTAL as the source, disable the PLL, change Master Clock to use IMO, and enable USB and have it use IMOx2.

Option B produces a more accurate clock than option A since it uses a crystal on the development kit for generating a clock signal, but both are sufficient for USB. Option B is also more power efficient than the internal oscillator, but in production it may be more desirable to save on the cost and physical space required for the crystal (see Figure 3).

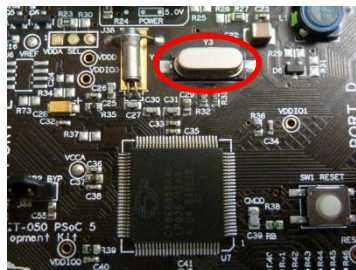


Figure 3: Location of the 24 MHz external crystal on the development kit

The PLL, which creates a faster or slower clock based on another clock, is unnecessary since both 48 and 24 MHz clocks are being produced for the USB and the master and bus clocks.

The default system clock settings has IMO run at 3 MHz with a +/- 1% error, which is too big an error for the USB clock. Using IMO at 24 MHz +/- 0.25% or XTAL fits within the requirements.

7. Also, change ILO to run at 100 kHz. Click Ok to confirm the changes.
8. Generate the application.

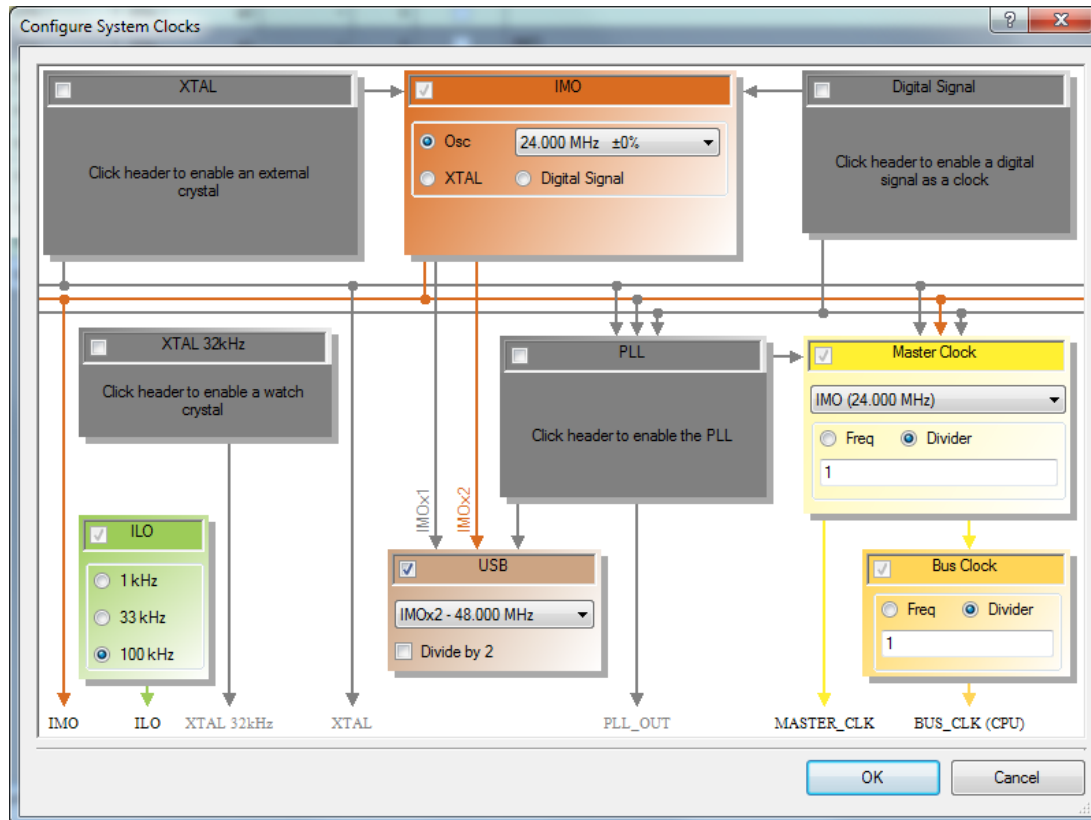


Figure 4: Configuring system clocks for USB (option A)

## Part 2: Programming the Firmware

### Outcome

The firmware will set up the USB device and simply read and echo back input from the host.

### Main

Because there is essentially only one way to properly initialize and continually read the USB UART, the code for the main function is fully provided in this part of the lab. See the code on the next page.

```
#include <project.h>

#define FALSE 0
#define TRUE 1

#define BUF_SIZE 64

int main()
{
    uint8 usbStarted = FALSE;
    uint8 usbBuffer[BUF_SIZE];
    uint16 usbBufCount;

    CyGlobalIntEnable;

    /* Start USBFS Operation with 3V operation */
    USBUART_Start(0, USBUART_3V_OPERATION);

    /* Start LCD */
    Display_Start();

    for(;;)
    {
        if(!usbStarted)
        {
            /* Wait for Device to enumerate */
            if(USBUART_GetConfiguration())
            {
                /* Enumeration is done, enable OUT endpoint for
                receive data from Host */
                USBUART_CDC_Init();
                usbStarted = TRUE;
            }
        }
        else
        {
            /* Check for input data from PC */
            if(USBUART_DataIsReady() != 0)
            {
                /* Read received data and re-enable OUT endpoint */
                usbBufCount = USBUART_GetAll(usbBuffer);
                if(usbBufCount != 0)
                {
                    /* Wait till component is ready to send more data */
                    while(USBUART_CDCIsReady() == 0);
                    /* Send data back to PC */
                    USBUART_PutData(usbBuffer, usbBufCount);
                }
            }
        }
    }
}
```

## Part 3: Running the Project

1. Build and load the project onto the PSoC development kit.
2. Either disconnect the USB cable from the development kit and plug it in to the USB port in the top middle, or use a second USB cable to connect the top middle USB port to the computer. This USB port is the one connected to the PSoC's USB device interface, while the other one is connected to the programming interface.
3. Windows will try and fail to install a driver for the USBUART interface.



The following instructions work with Windows 7. However, Windows 8 does not allow installation of unsigned drivers by default, and the driver produced by PSoC Creator is unsigned. Unfortunately, the only methods of getting around this require a reboot. The following URL leads to instructions for temporarily enabling installing unsigned drivers. This method undoes itself after another reboot, although unsigned drivers installed in the meantime are retained.

<http://www.makeuseof.com/tag/how-can-i-install-hardware-with-unsigned-drivers-in-windows-8/>

After enabling installation of unsigned drivers, the following directions work.

4. Open the Start menu, search for "Device Manager", and open it.
5. There should be an entry under Other Devices called "USBUART" with an exclamation point in its icon (Figure 5). Right click it and select "Update Driver Software..."

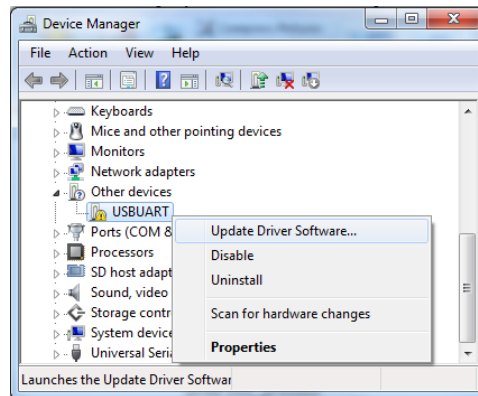
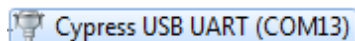


Figure 5: Finding USBUART in Device Manager

6. Choose "Browse my computer for driver software."
7. Browse to your workspace folder and choose the folder named Lab4.cysdn. Check the "Include subfolders" box and click next at the bottom of the window.
8. A warning will appear because the driver is unsigned. Ignore the warning and select "Install this driver software anyway."
9. The installation should complete successfully. Click Close.
10. If it didn't automatically select the new Cypress USB UART device, look in the Ports section of Device Manager to find the Cypress USB UART device. Take note of the COM port of the device. For example, the following device uses COM13.



11. Open your serial terminal software and connect to the COM port that belongs to the development kit. Here are directions for doing this with PuTTY (depicted in Figure 6).
  - a. Open PuTTY.
  - b. Change the connection type to Serial.
  - c. Change the serial line to the device's COM port.
  - d. Click Open.

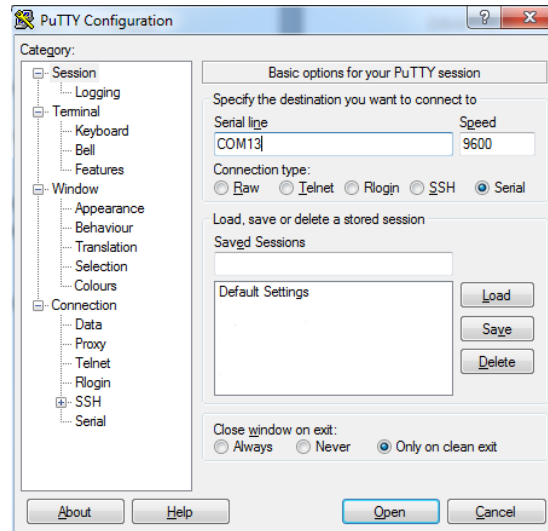


Figure 6: Opening the USB UART device in PuTTY

12. Type anything. Anything that is typed should appear in the terminal because the PSoC is programmed to echo back all input.

## Procedure Part B

### Outcome

In this part of the lab, a PWM component will control the brightness of an LED by quickly turning it on and off. The CMP value determines how bright the LED is because that influences how long the LED is on. The CMP value will be set by the user through the USB UART interface.

### Part 1: Designing the System

1. Reopen TopDesign.cysch
2. Add a Digital>Functions>PWM to the schematic area, name it PWM, set the period to 254, and set Interrupts under the Advanced tab to none. It should remain as the UDB 8-bit implementation with two outputs, CMP values of 127 and 63, and CMP types of less.



The period is set to 254 because the CMP type is less. If the period is 255, a CMP value of 255 will not make the LED 100% on, since the LED will be off when the period counter equals 255. Setting the CMP type to less or equal would fix this issue, but then a CMP value of 0 will not be completely off.

3. Add a new clock to the schematic, name it PWM\_clock, and configure it to be 12 MHz.
4. Connect PWM\_clock to the clock input of PWM.
5. Add a digital output pin to the schematic and name it LED4. Connect it to the pwm1 output of the PWM block.
6. Open Lab4.cydwr and configure LED4 to use port P6[3].
7. Generate the application.

## Part 2: Programming the Firmware

### Outcome

The firmware will change the period of the PWM to adjust the brightness of the LED. This period will be changed by the user through the USB UART interface. The user will enter a number from 0-255 and press enter to set the brightness of the LED.

### Initialization

To simplify the parsing of input from the USB UART, scanf will be used. Include stdio.h and inttypes.h to gain access to it. Add the following definitions (these will be referenced to later in this section):

```
#define CHAR_NULL '\0'
#define NUMBUF_MAX 3
#define CHAR_BACKSP 0x7F
#define CHAR_ENTER 0x0D
#define LOW_DIGIT '0'
#define HIGH_DIGIT '9'
```

And add the following variables to the rest of the variables at the beginning of the main:

```
char numBuf[NUMBUF_MAX+1] = "\0\0\0";
uint8 numBufCount = 0;
uint8 led4Bright = 0;
```

numBuf will store digits entered by the user over the USB UART, numBufCount will keep track of how many digits are entered, up to three (numBuf holds four for the ending null byte needed by scanf), and led4Bright will contain the current brightness of LED4.

Add PWM\_Start() after Display\_Start(). Set the initial brightness of LED4 using PWM\_WriteCompare1(led4Bright).

### Printing LED Brightness

Write a function to print the brightness of LED4 to the display. This function should be called directly after initialization (before the for loop) and whenever the brightness is modified. It should take the brightness as a parameter and have a void return type. Clear the display and print the following line to the display (0 is the current brightness):

```
LED4: 0
```

## Function for Putting Char to USB

Place the following function above the main function. It is necessary to wait for the USB device to be ready before placing characters into its buffer, so that part is better hidden behind a function.

```
void usbPutChar(char c)
{
    /* Wait till component is ready to send more data to the PC */
    while(USBUART_CDCIsReady() == 0);
    /* Send data back to PC */
    USBUART_PutChar(c);
}
```

## Modifying USB UART Code

The initialization code for USB UART (the GetConfiguration part) should not be changed.

The acquisition of data remains the same (the GetAll part). However, the code that echoes back the data (the if statement that has PutData inside it) will be replaced, so delete it.

After the GetAll, create a for loop to iterate over each character in usbBuffer based on the count stored in usbBufCount. For each character:

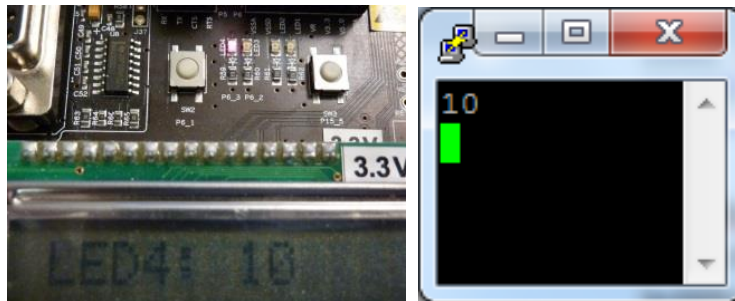
- If the character is between (inclusive) LOW\_DIGIT and HIGH\_DIGIT and numBufCount is less than NUMBUF\_MAX, add the character to numBuf at numBufCount, increment numBufCount, and call usbPutChar with the number.
- If the character is CHAR\_BACKSP and numBufCount is non-zero, decrement numBufCount, set the character at numBufCount to CHAR\_NULL, and call usbPutChar with CHAR\_BACKSP. Note that this works well with PuTTY, but other software may not erase the number that's deleted and just moves the cursor backwards one.
- If the character is CHAR\_ENTER and numBufCount is non-zero:
  - Convert numBuf to a uint8 and assign it to led4Bright using the following line of code. Note that a value greater than 255 will cause an overflow, but this will be ignored.  
`sscanf(numBuf, "%u" SCNu8, &led4Bright);`
  - Set the new brightness to the PWM controller using `PWM_WriteCompare1(led4Bright)`
  - Print the new brightness to the display using the function that was previously written
  - Call `USBUART_PutCRLF()` to move the cursor on the serial terminal to the beginning of the next line. Remember to wait until the USB component is ready before calling this function using:  
`while(USBUART_CDCIsReady() == 0);`
  - Reset numBuf by assigning all of its characters to CHAR\_NULL and set numBufCount to 0.
- If none of the conditions above are met, ignore them and do not call usbPutChar.

## Part 3: Running the Project

1. Build and load the project onto the PSoC development kit.
2. Plug the PSoC's USB interface into the computer and open the serial terminal.
3. Test the program to make sure it works:
  - Starting out, the LCD display should say "LED4: 0" and LED4 should be off.



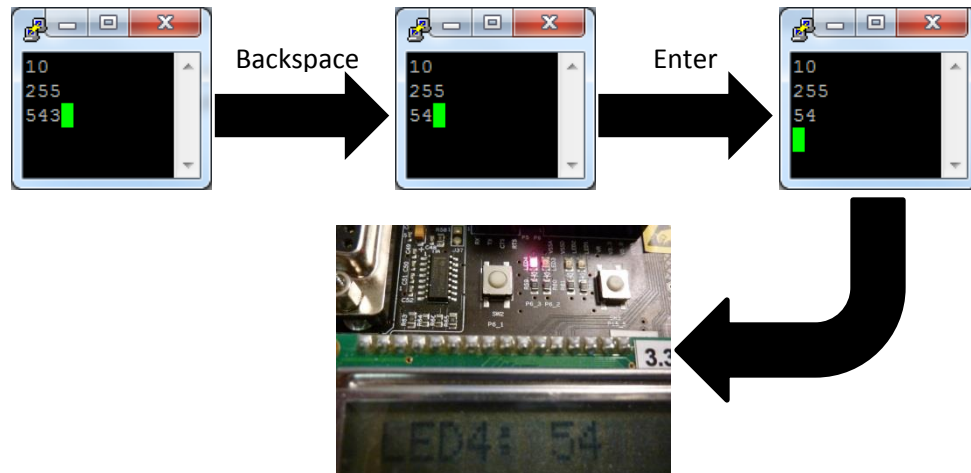
- Type 10 into the terminal. The number should appear in the terminal. LED4 and the LCD display should remain unchanged.
- Press enter. LED4 should appear dim, the LCD display should say "LED4: 10" and the cursor on the terminal should move to the beginning of the next line.



- Type 255 into the terminal and press enter. The LCD display should now say "LED4: 255" and LED4 should be bright.



- Attempt to type more than three digits into the terminal. There should be no more than three digits echoed.
- Attempt to type any letter. Nothing should be echoed.
- Press backspace. The last digit should be erased and the cursor should move back one space.
- Press enter. The LCD display should be updated with the two digit number in the terminal (instead of the three digit number that was originally there, or a number different from what was entered if the three digit number was greater than 255).



- Enter a three digit number and press backspace three times. There should be no more digits on the line.
- Press enter and backspace. Nothing should happen since there are no digits entered.