

# Controlling a PSoC Big Board from a PC Web Browser Using a Serial Link

Veloria Pannell, Eric Ponce

August 2024

## 1 Overview

You can use a PC Browser, like Edge or Chrome, to create a graphical user interface (GUI) and control a microcontroller, like a PSoC, with a serial connection. This tutorial will show you how to configure the PSoC Big Board (CY8CKIT-050 PSoC 5LP Development Kit) to communicate with an HTML webpage using Chrome's Web Serial API. This tutorial covers the basic hardware, firmware, and software components required for an example webpage that allows students to input data via interaction with an HTML webpage in order to display text on an LCD and alter the brightness of an LED.

The hardware refers to the on-board wiring and cypress schematic components, the firmware includes a C file in Creator, and the software includes any other code outside the Creator code (e.g. HTML, CSS, and JavaScript). A PSoC Creator 3.3 project, as well as the HTML, CSS, and JavaScript files can be found on the course website. Additional resources for more information about the API and its usage are provided at the end of this document.

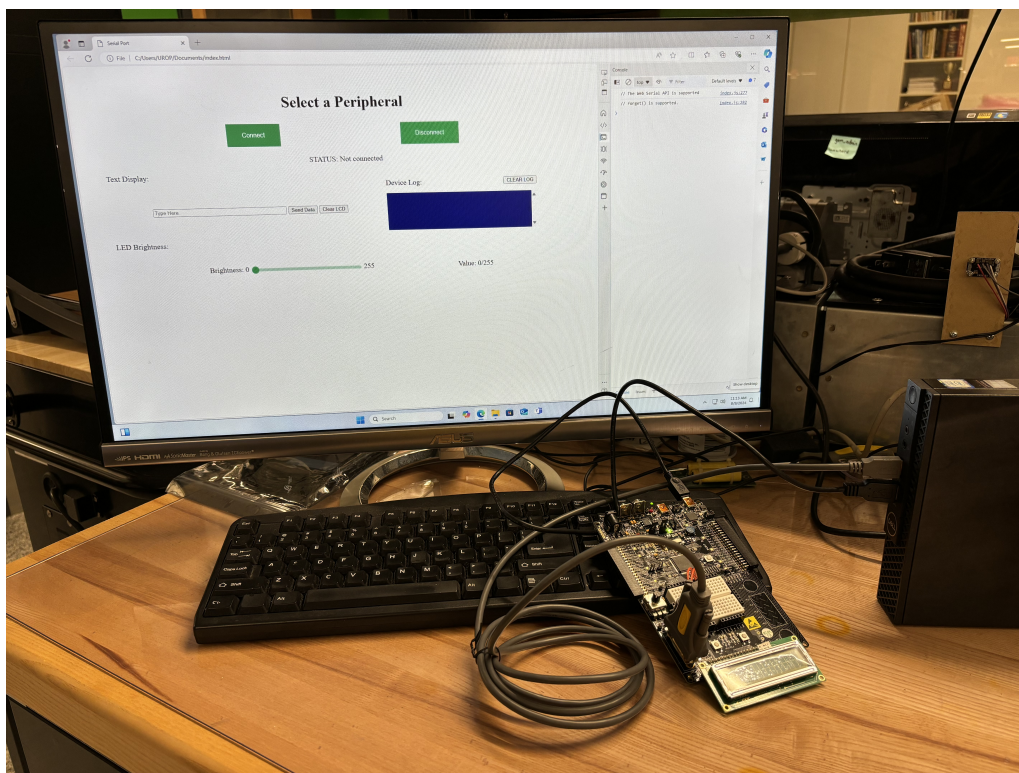


Figure 1: PC connected to PSoc Big Board

## 1.1 How it works

Here is a basic overview of how to use the webpage; a more detailed version is located later in the manual.

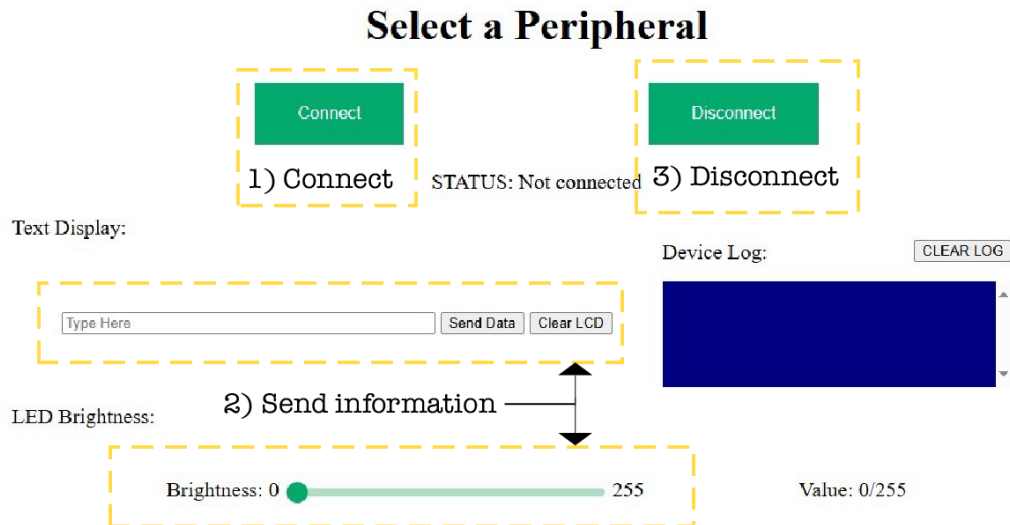


Figure 2: The webpage

- Two buttons are provided to establish/disestablish a serial connection with the PSoC Big Board.
- The text box can be used to display text on the LCD
- The slider can be used to adjust the brightness of LED4 on the PSoC "Big Board" Evaluation Kit.
- Feedback from the PSoC appears in the Device Log; the Device Log and LCD can be cleared with the appropriately labeled buttons.

Now that you know how it works, let's build it!

## 2 Hardware

You will need your PSoC 5LP Big Board, the compatible LCD, a mini-USB cable, and a USB-to-RS232 Kable, both of which will be connected to a PC's USB ports. The mini-USB cable provides power to the board while the USB-to-RS232 Kable is for serial communication. Because we are using the LCD, ensure the jumper is set to power your board with 3.3V. The PSoC Board setup and Creator components and configurations are shown in more detail below (figures 3-5).

### 2.1 PSoC Board and Creator

Place the LCD in its designated ports at the bottom of the board. Then, connect Rx\_1 and Tx\_1 to unused ports with wires; port selection is arbitrary but must match the assignments in the .cydwr file. Here, P0[7] and P0[6], respectively, are used. For the other components, the LED is assigned to P6[3] and the LCD to P2[6:0].

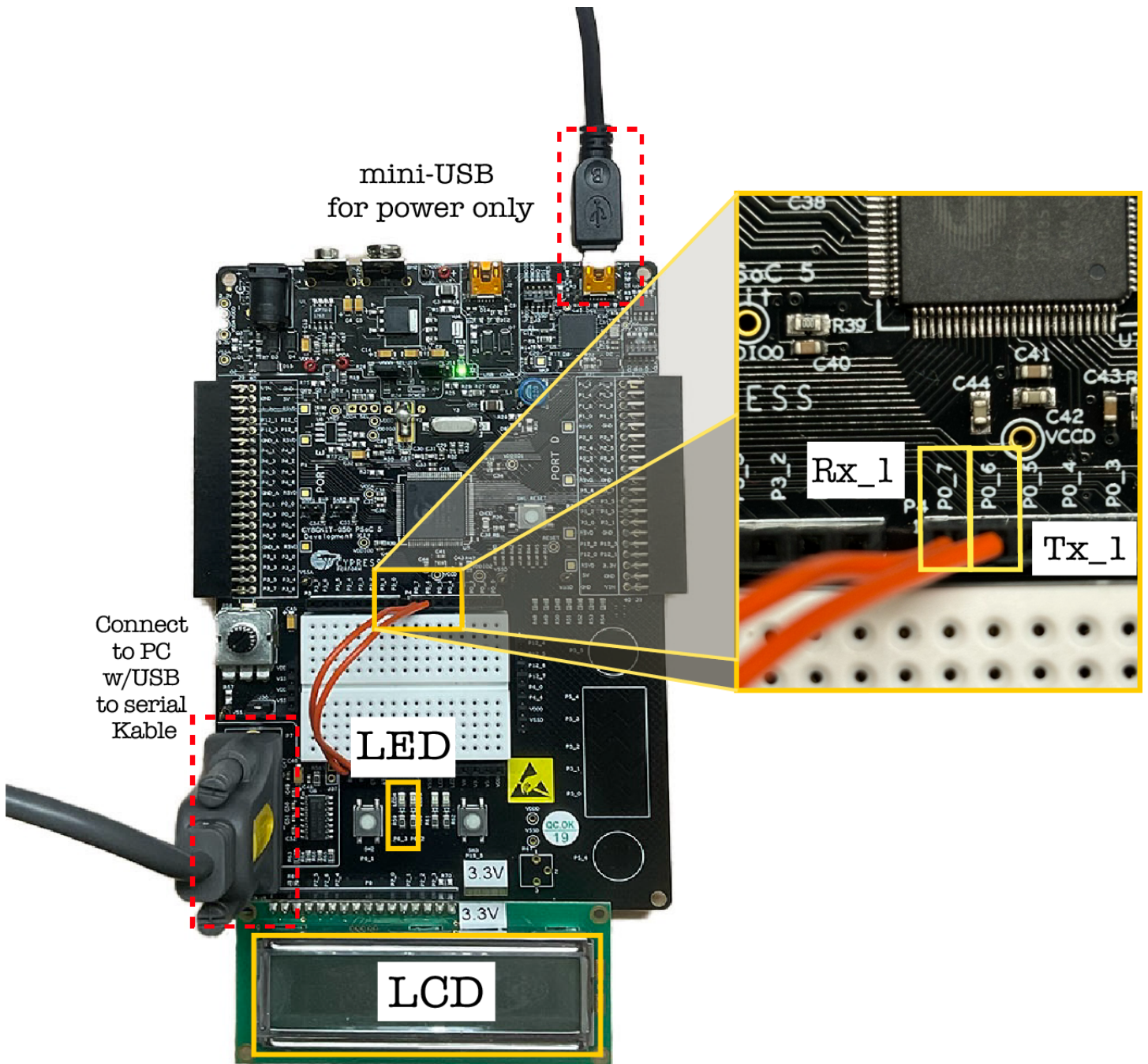


Figure 3: PSoC Board Setup

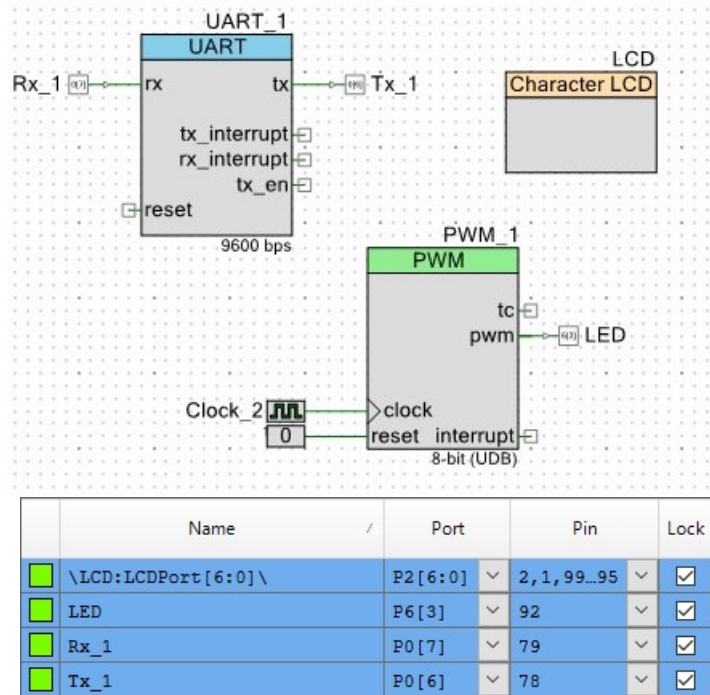


Figure 4: Example PSoC Creator Top-level Schematic and Pin Editor

For the Creator schematic, the required components blocks include a UART, PWM, clock, and LCD, as well as a digital input pin for Rx\_1 and digital output pins for Tx\_1 and an LED.

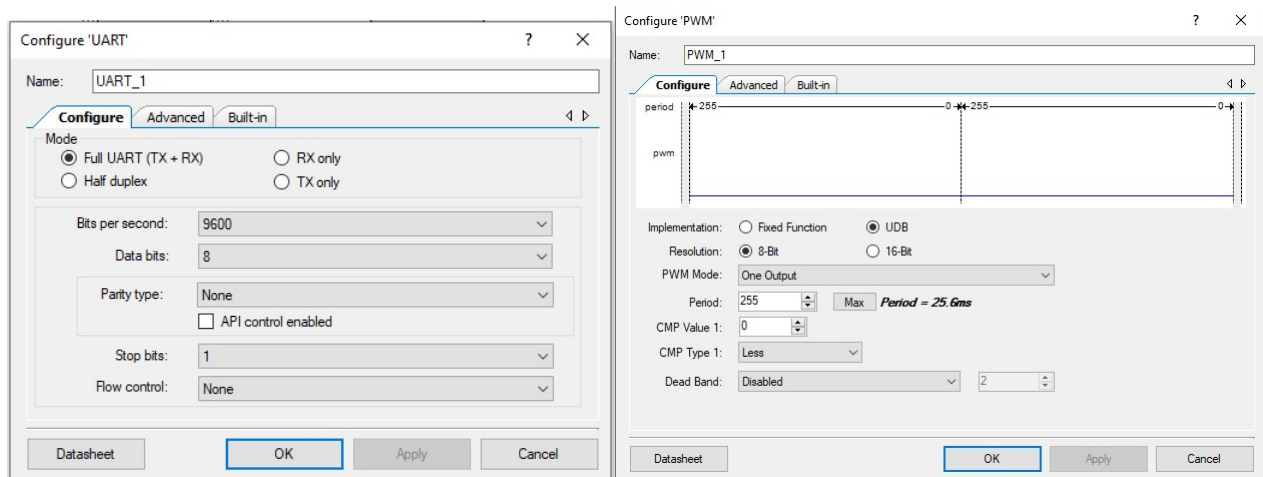


Figure 5: UART and PWM Configurations



### 3 Firmware

If you navigate to `main.c`, you will see that we are sending and receiving data using Creator’s UART block and making API calls to the PWM and LCD according to the user’s input. Upon device reset (either after finishing programming or pressing the SW1 Button), the components are initialized, the UART transmits “ \*Device Reset\* ” to print in the Device Log, and the LED is turned off. After this initial setup, the UART awaits more data from the PC.

The variable `ch` holds the most recently received char from the API call `UART_GetChar()` . In order to decide how to handle each received `ch` , special string codes are sent to tell the PSoC what to do. In the provided Creator code, the 4-char string codes include “CLR ”, “LED ”, and “TXT ”, stored as the macros `CLEAR` , `LED` , and `TEXT` . These codes are dealt with in the helper function, `parse()` . There is also a variable called `decode` that determines whether to handle received data as text or an LED brightness. `decode` will be set to 1 when it receives `TEXT` and set to 0 when it receives `LED` ; `decode` is 1 by default.

#### 3.1 Printing to the LCD

When `decode` is 1, incoming data is handled as text. Clicking the “Send Data” button will send the user’s text input, and valid ASCII characters that are received are both printed on the Big Board LCD and returned as an echo displayed on the webpage Device Log.

In the Creator code, the PSoC displays each `ch` to the LCD with `LCD_PutChar(ch)` , before incrementing the LCD cursor position and then sends the `ch` back as acknowledgement to the PC with `UART_PutChar(ch)` .

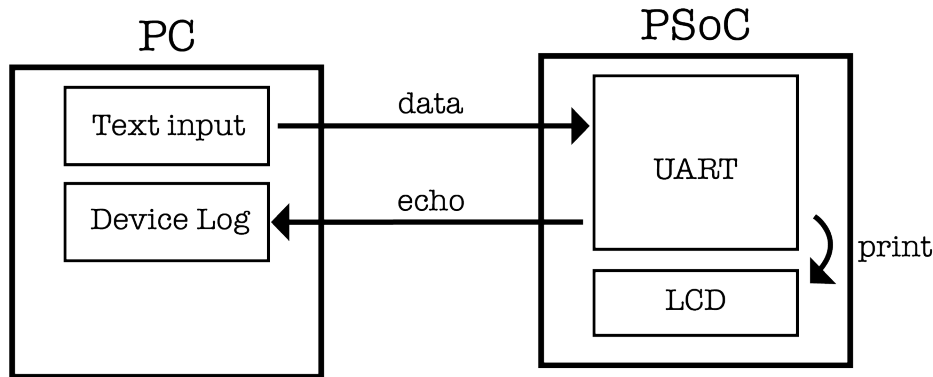


Figure 6: Text sent by the user displays on the LCD; the echo appears in the Device Log

### 3.2 Clearing the Display

When the “Clear LCD” button is pressed, the macro `CLEAR` is sent to the PSoC, which clears the LCD and transmits “\*Display Cleared\*” to the webpage Device Log.

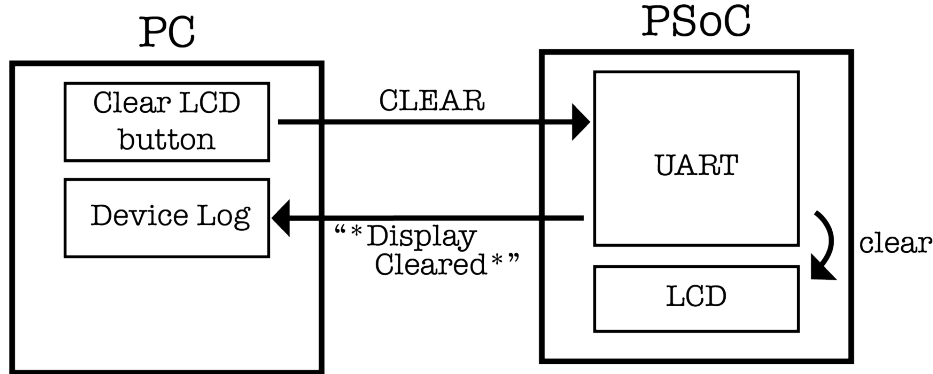


Figure 7: Device communication for clearing the LCD

### 3.3 Adjusting LED Brightness

Interacting with the slider sets `decode` to 0 with the `LED` macro in order to mark that the following inputs are to be interpreted as an LED’s brightness. Once you stop using the slider, `decode` is set back to 1 with the `TEXT` macro.

Rather than printing the brightness values to the LCD and Device Log, the PWM component updates its compare value to the received value with `PWM_WriteCompare()`. This updates the PWM component’s duty cycle and as a result, adjusts the perceived brightness of the LED. The ASCII values in the `TEXT` macro are reserved for returning to reading text, so they do not affect the LED value handling.

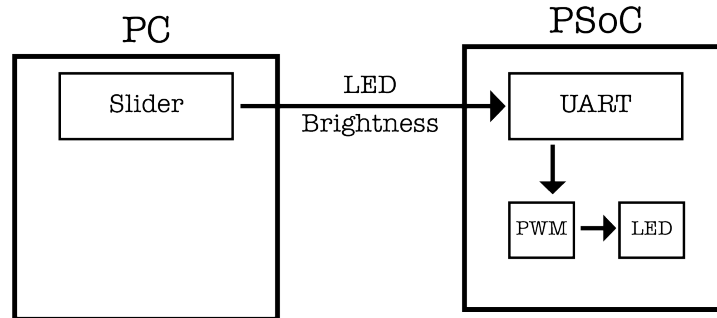


Figure 8: For each LED value sent with the slider: (1) `LED` sets `decode` to 0 (2) Input changes duty cycle (3) `TEXT` sets `decode` back to 1

## 4 Software

For the software, we have three main files that together create this webpage. The HTML file is the skeleton of the webpage, providing structure and rendering everything that we see, while the JS file provides functionality and allows HTML elements to interact with users or other elements. The CSS file just packages it all together and makes it look nicer, though it is not necessary to function.

This webpage uses a web API (application programming interface) to communicate with the UART via the serial port. **The API is available on all desktop platforms (ChromeOS, Linux, macOS,**

and Windows) in Chrome 89, but at the time of writing is limited to browsers Chrome, Edge, and Opera. The HTML and JavaScript files are thoroughly commented, print updates in the console, and can be customized for different devices and functions (details in ‘Customization’ section). Besides a brief overview of the API’s function and usage, more resources are linked at the end of this document.

## 4.1 Code and APIs

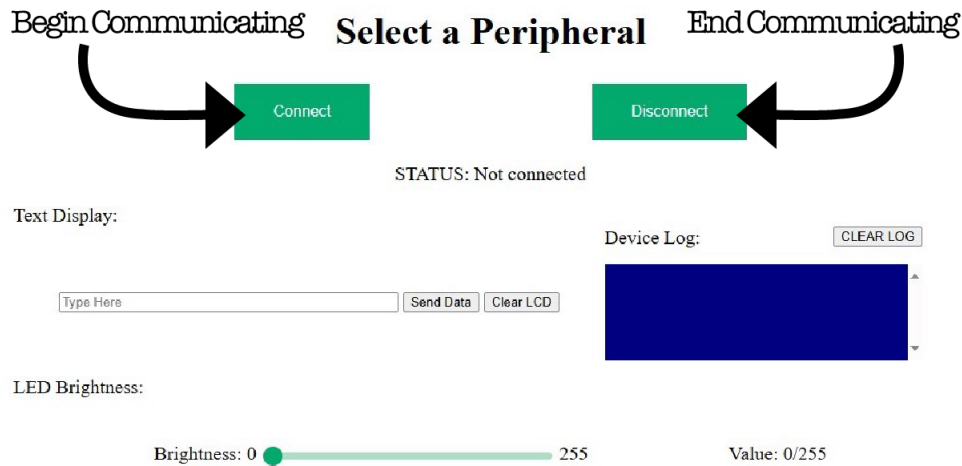
The Web Serial API relies on a Streams API that can create instances of Readable and Writable Streams through which data can be exchanged. The JavaScript code involves 5 main asynchronous helper functions that use the API: `connect()`, `textRx()`, `textTx()`, `ledTx()`, and `disconnect()`.

To connect to a device, a serial port object is created; when initializing the port, a Readable Stream and a Writable Stream are created to form a bidirectional communication pathway with a device. Then a reader object is locked to the stream to continuously listen for data (Note: only one stream can be read from or written to at a time).

To write data, a writer object is locked to the stream, data is written, and the writer releases its lock, opening the Stream back up to other writer objects to write new data. Finally, to disconnect, both the reader and writer must release any remaining locks to the Stream before the port can be closed.

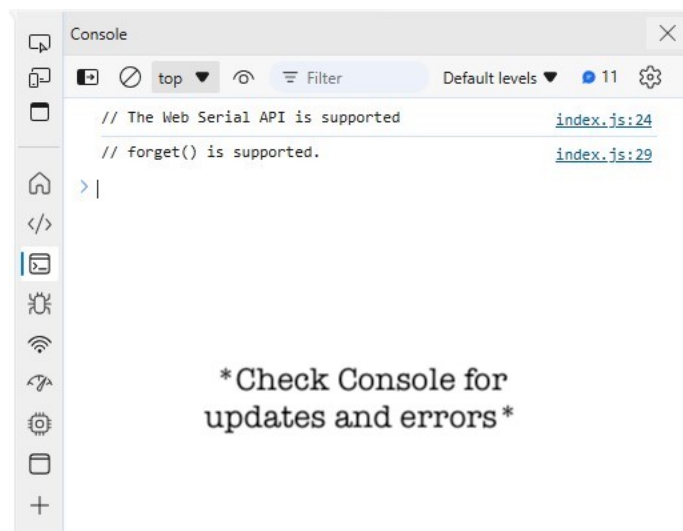
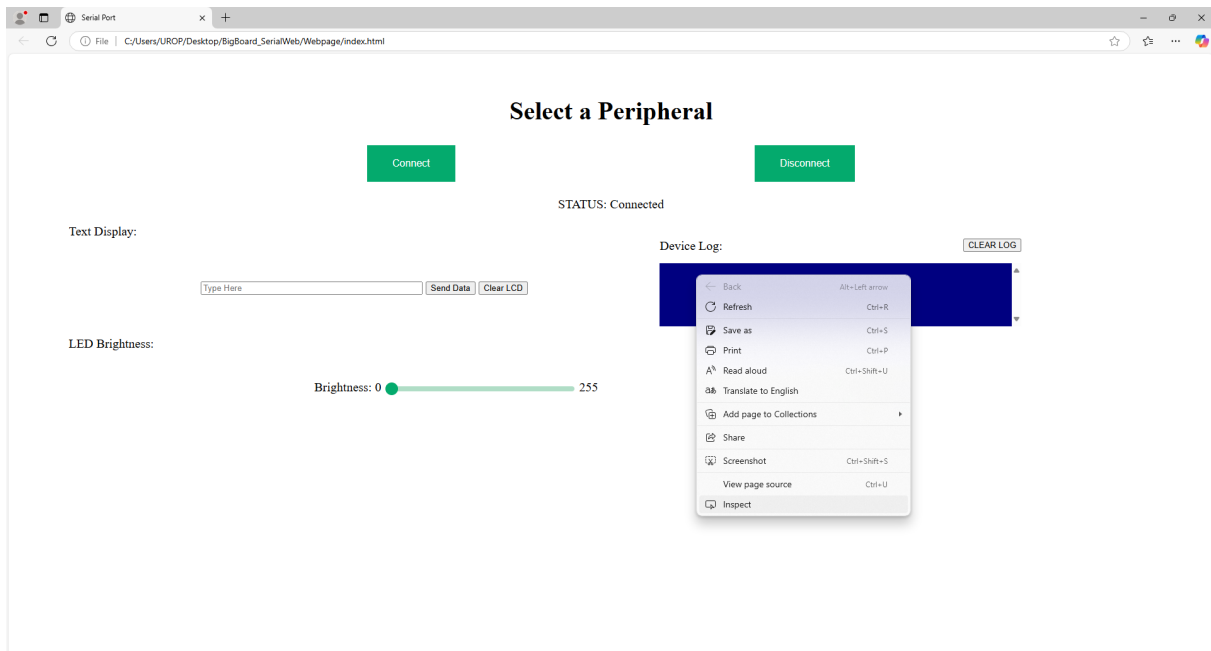
## 5 Using the Webpage

Selecting the green ‘Connect’ and ‘Disconnect’ buttons will begin and end communication with a peripheral device. When you connect, you will be prompted to select an available device. The web page is programmed to detect and display all connected COM ports and will connect to the user’s selected COM port at a baud rate of 9600, but these settings can be customized (detailed later in ‘Customization’ section).



Anytime `console.log()` is used in the JS file, the contained data is printed in the webpage console; this is like the webpage version of a `print` statement and can be useful for debugging. To view the console, navigate to the browser’s debug tools: First, right-click and select ‘Inspect’ and view the ‘Console’ tab. The ‘Source’ tab in the same menu also conveniently displays any source files, notably the HTML, CSS, and JS code. [F12] can also be used to access this menu.





## 5.1 Printing to LCD

To print to the LCD, type text into the input box that says ‘Type Here.’ When you click the ‘Send Data’ button, your text will be displayed on the LCD as well as in the Device Log (example below). Clicking ‘Clear LCD’ erases the text displayed on the LCD; ‘CLEAR LOG’ empties the Device Log.

Text Display:

6.115 rocks!!! Send Data Clear LCD

Device Log:

CLEAR LOG



Text Display:

Type Here

Send Data

Clear LCD

Device Log:

CLEAR LOG

6.115 rocks!!!



## 5.2 Adjusting LED Brightness

To adjust the brightness of the LED, scroll through the brightness slider; the values, displayed on the webpage to the right of the slider, will dim/brighten LED4 on the PSoC. In the figure below, the left image shows a value of 20, and the right image shows a value of 255 (right).

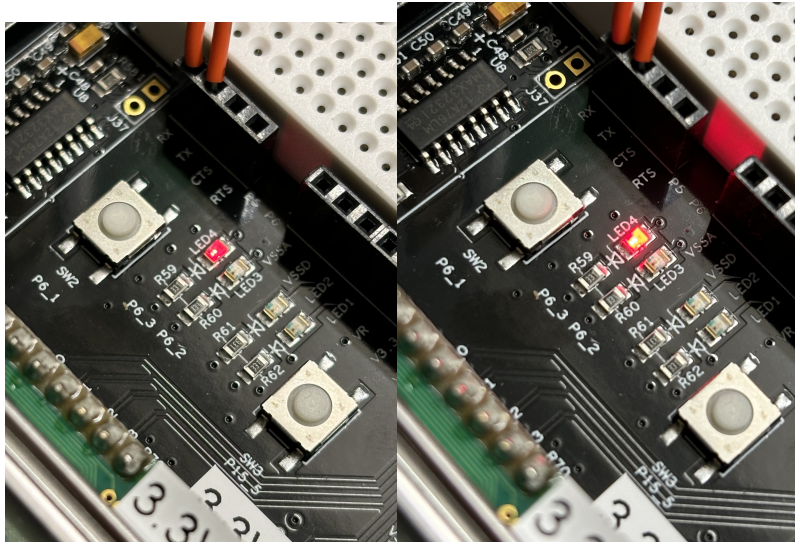
LED Brightness:

Brightness: 0



255

Value: 0/255



## 6 Customization

This section provides guidance for modifying the software to accommodate other ports, baud rates, and also covers webpage customization.

## 6.1 Detecting Devices

The webpage can detect all connected COM ports; in order to detect only specific ports, alter the array of product filters in the Javascript file, and insert ‘{filters}’ as shown below.

```
54  /* CONNECT() */
55  async function connect() {
56    try {
57      // Prompt user to select any serial port. [option to add filters]
58      // ADD FILTERS: port = await navigator.serial.requestPort({ filters });
59      port = await navigator.serial.requestPort();
60      const { usbVendorId, usbProductId } = port.getInfo();
61    } catch (err) {
62      console.log("Error on request: ", err);
63    }
64  }
```

You will need to know the usbVendorId and, optionally, a usbProductId.

## 6.2 Setting a Different Baud Rate

Currently, the port is automatically opened with a baud rate of 9600; this can be changed to any other value (e.g., 115200) to match the peripheral. For an even more general option, the user can be prompted with a popup with the `prompt()` command.

```
132  try {
133    //set baud rate to user's input. [option to prompt user for custom baud rate or specific bps]
134    //CUSTOM BAUD RATE: await port.open({ baudRate: prompt("Baud rate") });
135    //SPECIFIC BAUD RATE: await port.open({ baudRate: ### });
136    await port.open({ baudRate: 9600 });
137    const { usbVendorId, usbProductId } = port.getInfo();
138  } catch (err) {
139    console.log("Error on open: ", err);
140  }
```

## 6.3 Adding Functional Components

If you want to edit these files to create your own Webpage, this section will provide a simple process to add basic interactive elements to the webpage. More details on the HTML, CSS, and JS code is in the Appendix, and check out MDN Web Docs for extensive documentations (in Helpful Links section).

**1. HTML Element** The first thing you need to do is create the visual element that appears on the webpage that you can interact with. This means adding tags to insert a component with your desired formatting. Give it a unique id that you can use to reference in the Javascript file.

- To add a button, use `<button>Press Me</button>` to add a button that says “Press Me.” Customize this text to what you want the button to say. If you simply want to call a function after clicking the button, add the ‘onclick’ parameter with `<button onclick = “function()”>`. Otherwise, its function will be handled in JS.
- To add a text input, use `<input type=“text” />`. You can add a placeholder parameter if you want faint text when there is no input (like the “Type Here” for the original webpage). Adding a size parameter will change the width of the input from the default 20 characters.
- To add a slider, use `<input type=“range” />`. Adding min and max parameters changes the range of inputs, and value will set the default input.  
Example: `<input type = “range” min = “[LO value]” max = “[HI value]” value = “0” />`

**2. JS Function** After you have an HTML element for the component you want to add, you can make it do something. You can use its unique id to add an “eventListener” to “listen” for an event (like a click) on that element and run specific code when it occurs. For common events like click or input, you can also make an `onInput()` function or an `onClick()` function, which are specific to those events.

Key pieces of information:

- **id:** use this to refer to the HTML element – `document.getElementById("[id]");`  
Store in a variable if you would like (i.e., `button = document.getElementById("[button's id]");`)
- **event:** this is “click” for a click. Other examples: “dblclick” for double click, “mousedown”/“mouseup” for pressing/releasing the mouse, “mouseover” for hovering, “input” for each new input value, “focus”/“blur” for engaging/disengaging with the element, etc.
- **code:** this is what you want to execute once the event is detected. If you don’t want to define a separate function, you can use an arrow function instead – `(param) => {code...}`
- **Some Examples:**
  - `document.getElementById("[id]").addEventListener("event", function)`
  - `button.addEventListener("click", (ans) => {console.log(ans);})`
  - `box.addEventListener("event", async () => {await function();})`
  - `document.querySelector("#[id]").addEventListener("input", (e) => {Tx(e); console.log(e);})`
  - From the serial webpage (sending LED values):

```
24 // Slider for LED
25 const slider = document.getElementById("bar"); // HTML element w/ ID of "bar"
```



```

190  /* SENDING LED */
191  // On any change in slider value, the value is sent to PSoC
192  slider.oninput = async function () {
193      //purely update the html text
194      LEDvalue.innerHTML = "Value: " + this.value + "/255";
195
196      // Tell PSoC to interpret as LED value
197      if (encode) {
198          encode = false;
199          await textTx(ledUpdate);
200      }
201      // write the slider value to serial port
202      const uint8 = new Uint8Array(1);
203      uint8[0] = this.value;
204      await ledTx(uint8);
205  };
206  // Upon releasing LED slider
207  slider.addEventListener("mouseup", async (e) => {
208      // Tell PSoC to interpret as an LED value
209      if (encode) {
210          encode = false;
211          await textTx(ledUpdate);
212      }
213      // send LED value to serial port & update html text
214      const uint8 = new Uint8Array(1);
215      uint8[0] = slider.value;
216      ledTx(uint8);
217      LEDvalue.innerHTML = "Value: " + slider.value + "/255";
218      // Return to default text state after releasing slider
219      encode = true;
220      await textTx(txtUpdate);
221  });

```

## 7 Appendix

There are many online resources and documentation to learn about any concepts for HTML, CSS, and JS, but these appendices will just cover the concepts that were utilized for this webpage specifically. For more help, check out MDN Web Docs (in Helpful Links section)

### 7.1 HTML/CSS

The code used in the HTML file can be divided into 3 different categories: functionality, formatting, and interactive elements.

**Function:** With the `<link>` and `<script>` tags, shown below, the CSS and JS files are connected to the HTML, giving the basic webpage some visual styling and fundamental functionality.

```
7      <!-- CSS (Cascading Style Sheet) file is separate but linked here -->
8      <link rel="stylesheet" type="text/css" href="index.css" />

72     <!-- JavaScript file is separate but linked here -->
73     <script src="index.js" type="text/javascript"></script>
```

**Format:** The `<title>` tag sets the text that appears on the tab, and everything under the `<body>` tag is a visible element, such as a text element. This HTML file only uses one type of header ( `<h1>` tag), as well as a paragraph element ( `<p>` ) tag.

Further organization of the HTML document utilizes the `<div>` and `<span>` tags, which are both generic content dividers, separating the document into different sections; divs stack vertically while spans stack horizontally.

## This is a heading inside a div element

This a paragraph inside a div element.

## This is a heading inside a div element

This a paragraph inside a div element.

This text is inside a span element. This text is inside another span element.

\*Example from article on div vs span (blog.hubspot.com)

**Interaction:** The interactive elements in this HTML document include the `<button>` tag, a simple button, and the `<input>` tag to get user input, customized with the 'type' parameter. For this webpage, the input text box ( `type="text"` ) and the slider ( `type="range"` ) are both `<input>` elements. Any HTML elements can be given a 'class' or a unique 'id', which can be used to reference elements in a different file.

In order to program what these elements do, you can use an `eventListener` (described in more detail under JS appendix), or if just simply calling a function, you can use a parameter associated with a specific event to call that function. For example, with the 'Connect' button, it has an `onclick` parameter that is set to the function `connect()` .

```
14      <!-- First row for connection/disconnection; triggers functions in JavaScript file -->
15      <!-- Functions in JS: -->
16      <div class="row">
17          <button class="button" id="connect" onclick="connect()">Connect</button>
18          <!-- function connect() in JavaScript file -->
19          <button class="button" id="disconnect">Disconnect</button>
20          <!-- "Click" eventListener in JavaScript file -->
21      </div>
```

The LED input slider is customized with pretty self-explanatory parameters and labeled with `<label>` tags.

```

59      <!-- slider input on HTML end -->
60      <span>
61          <label for="PWM">Brightness: 0</label>
62          <input type="range" min="0" max="255" name="PWM" id="bar" value="0" />
63          <label for="PWM">255</label>
64      </span>

```

The slider’s `name` parameter allows the labels to be paired with it through their `for` parameters matching “PWM.” As you can probably tell, `min` is the lowest value input of the slider, and `max` is the highest. `value` sets the initial slider value upon rendering the webpage.

The CSS for this project is not necessary to the function of the webpage but adds visual appeal. Since the code just applies basic adjustments to sizing and color, paired with the comments already in the CSS file, this section will not cover much CSS since it is mostly self-explanatory.

Besides basic sizing and spacing, the 1-dimensional layout method of Flexbox was used to center and organize sections of the webpage (more information on MDN Web Docs). It sorts content into rows or columns and streamlines the placement of the elements within each.

HTML elements can also be specifically referred to by class with `.` (e.g., `.button{ }`), by id with `#` (e.g., `#main-text{ }`), or by type of element (e.g., `input[type="range"]`). In CSS, the anatomy of the ‘range’ input includes a “thumb” portion (the circle), as well as a “track” portion to slide the thumb along, and was visually customized in this CSS file. The buttons were made to change colors when the mouse hovers over with `.button:hover{ }`.

## 7.2 JS

The Javascript file contains several functions and pairs them, along with any other code, to ways the user interacts with HTML elements, and in the case of the Web Serial API, with the browser and serial connection as well. `console.log( )` is used throughout the file to print text to the console for debugging and user feedback purposes.

The variables in this file are declared with `var` (allowed to change) and `const` (do not change), and they store the string codes ( `CLEAR` , `TEXT` , and `LED` ), HTML elements (e.g., buttons, text, slider), and any objects related to the serial communication API. HTML elements can be referred to by ID with `getElementById("[ID]")` or more generically with `querySelector("[attribute]")`. Using a `#` refers to id, so `getElementById()` can be used interchangeably with `querySelector("#[ID]")`. Any variables storing something beginning with `navigator.serial` are from the Web Serial API.

The two main methods here to program these interactions are the EventTarget interface (eventListeners) and the `async function` declaration.

To use the EventTarget interface, code is paired with an “event,” such as `"click"` or `"mouseup"`, similar to using the HTML parameter `onclick` but more organized for increased amounts of code and the ability to add console print statements. The syntax would look something like:

```

[HTML ELEMENT].addEventListener("event", function())
or
[HTML ELEMENT].addEventListener("event", () => { CODE } )

```

The associated code above is put in an arrow function, which is useful for inlaying a function with several lines of code rather than calling a single, separately defined function, such as `connect()` (more information on MDN Web Docs).

The five helper functions in this JS file that *are* separately defined (i.e., `connect()` , `disconnect()` , `textTx()` , `ledTx()` , & `textRx()` ) use the `async` and `await` syntax (more information on MDN Web Docs). They are initialized as asynchronous functions, allowing them to use the keyword `await` to wait for a line of code to completely finish being run, returning a Promise, before moving onto the next line. This usage of `async/await` is critical to the function of the Web Serial API, as it is fundamentally embedded in the API itself and better accommodates the entire serial communication process.

## 8 Helpful Links

- Chrome Developer Docs – <https://developer.chrome.com/docs/capabilities/serial>
- Streams API Guide – [https://developer.mozilla.org/en-US/docs/Web/API/Streams\\_API/Concepts](https://developer.mozilla.org/en-US/docs/Web/API/Streams_API/Concepts)
- Web Serial API Guide – [https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Serial\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Serial_API)
- Web Serial API Docs – <https://wicg.github.io/serial/>
- UART datasheet – [https://www.infineon.com/dgdl/Infineon-Component\\_UART\\_V2.50-Software%20Module%20Datasheets-v02\\_05-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e7fc71181](https://www.infineon.com/dgdl/Infineon-Component_UART_V2.50-Software%20Module%20Datasheets-v02_05-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e7fc71181)
- LCD datasheet – [https://www.infineon.com/dgdl/Infineon-Component\\_Character\\_LCD\\_\(CharLCD\)\\_V2.20-Software%20Module%20Datasheets-v02\\_02-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e796a7d09d6](https://www.infineon.com/dgdl/Infineon-Component_Character_LCD_(CharLCD)_V2.20-Software%20Module%20Datasheets-v02_02-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e796a7d09d6)
- PWM datasheet [https://www.infineon.com/dgdl/Infineon-Component\\_PWM\\_V2.20-Software+Module+Datasheets-v03\\_03-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e801c7611bd](https://www.infineon.com/dgdl/Infineon-Component_PWM_V2.20-Software+Module+Datasheets-v03_03-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e801c7611bd)
- MDN Web Docs – <https://developer.mozilla.org>