CURIE Academy, Summer 2021
Lab 2 Notes: Computer Engineering – Software Perspective

Prof. Christopher Batten
School of Electrical and Computer Engineering
Cornell University

The field of computer engineering is at the interface between hardware and software and seeks to balance the tension between application requirements and technology constraints. In Lab 1, you explored the field of computer engineering from a hardware perspective by assembling basic logic gates to implement a simple "calculator" for adding small binary numbers. In this lab, you will explore the field of computer engineering from a software perspective by incrementally programming a microcontroller in C++ to implement an IoT "smart light" system. These lab notes provide a brief survey of background information relevant to understanding the purpose and context for the lab.

As illustrated in Figure 1, computer systems can be viewed as a stack of abstraction and implementation layers from applications at the highest layer to technology at the lowest layer. In these lab notes we will briefly discuss the application, algorithm, programming language, operating system, compiler, and instruction set layers as they relate to our "smart light" system. In the actual lab session, you will have an opportunity to put what you have learned into practice. We will focus on some layers more than others, but by the end of this lab you should have a good understanding of how computer engineers can leverage these layers to design software for future computing systems.

1. Application: Smart Light

Figure 2 illustrates an example "smart light" system from a company called Brilliant. An internet-connected light switch enables turning on lights anywhere in the home, but this same light switch can

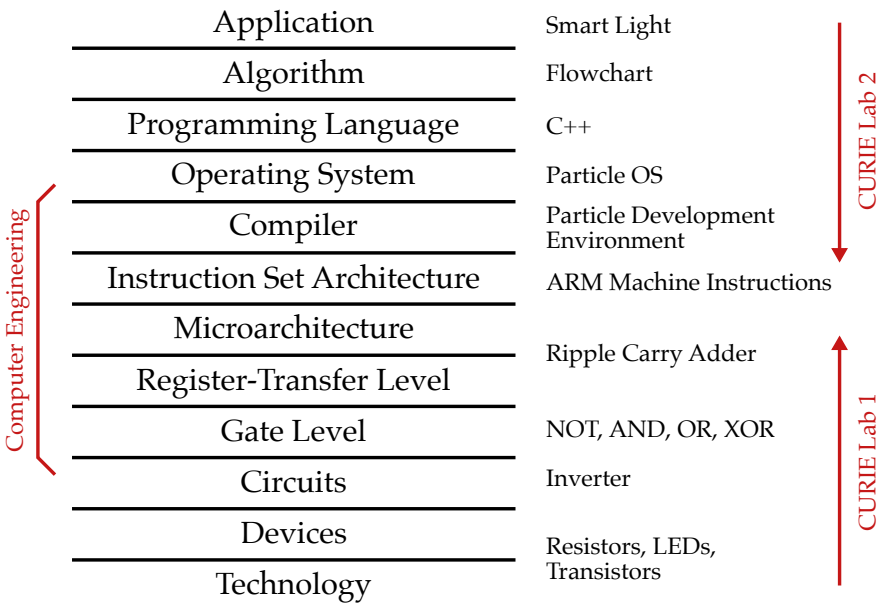


Figure 1: Computer Systems Stack

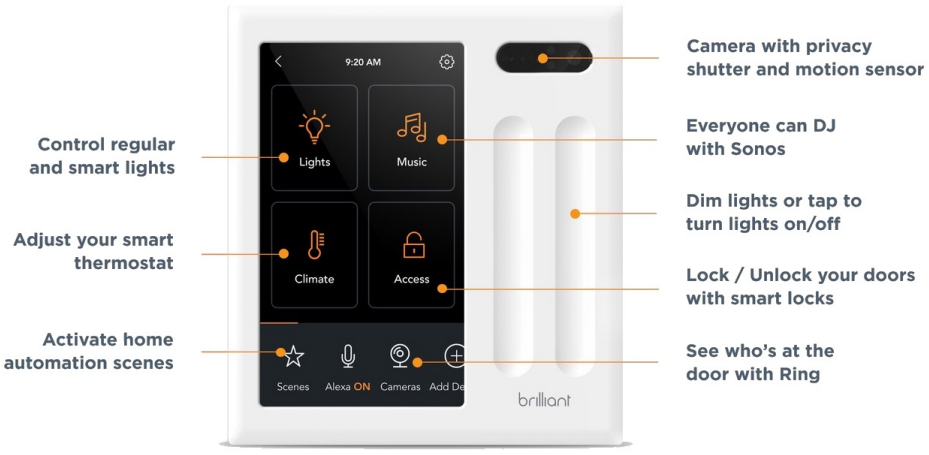


Figure 2: Example Commercial "Smart Light" System from Brilliant

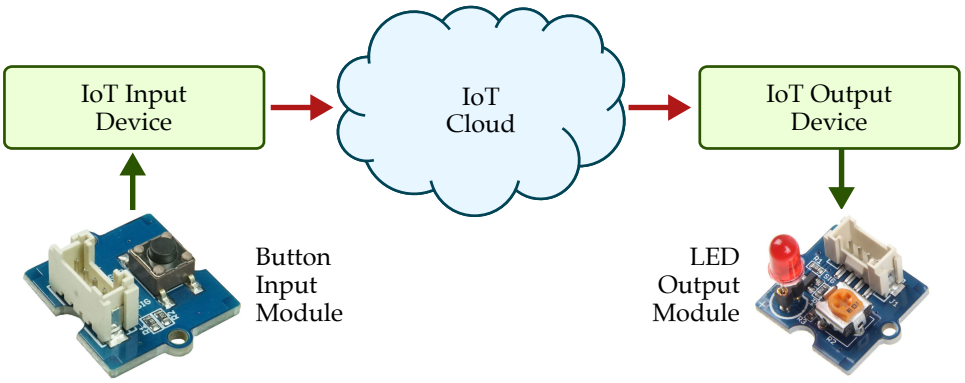


Figure 3: Diagram of Simple "Smart Light" System

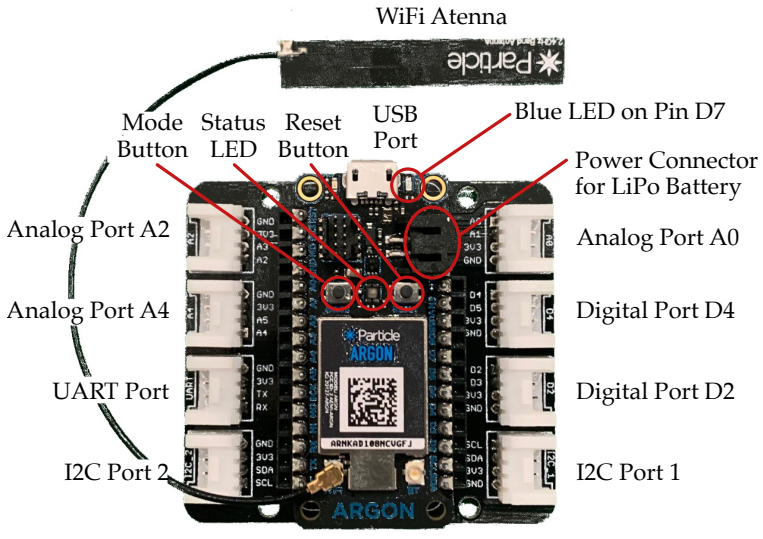


Figure 4: Particle Argon

also adjust the thermostat, lock/unlock doors, monitor security cameras, and control whole-house music systems.

Figure 3 shows the simple "smart light" *application* we will building in this lab. A button input module is attached to the IoT input device which will send a message to the IoT cloud whenever the button is pressed. An LED output module is attached to the IoT output device which will receive messages from the IoT cloud and turn on the LED whenever the button is pressed. The IoT input device with a button input module is a simple version of more sophisticated smart light switches, and the IoT output device with an LED output module is representative of a "real" smart light bulb.

Figure 4 shows a Particle Argon, which is the actual IoT device we will be using in this lab. The Particle Argon includes a small processor along with a WiFi communications subsystem. The Particle Argon can be powered either through a USB port or a special connector for a LiPo battery. Right in the middle of the Particle Argon is multi-colored status LED which is the primary way to determine what the Particle Argon is doing. Here is the meaning of the most common status LED signals:

- Breathing Cyan: Argon is connected and operating normally
- Blinking Green: Argon is connecting to WiFi network
- Blinking Cyan: Argon is connecting to Particle cloud
- Blinking Blue: Argon is in listening mode
- Blinking White: Argon's WiFi module is off
- Breathing Magenta: Argon is in safe mode
- Rainbow Colors: Argon is being signaled from cloud

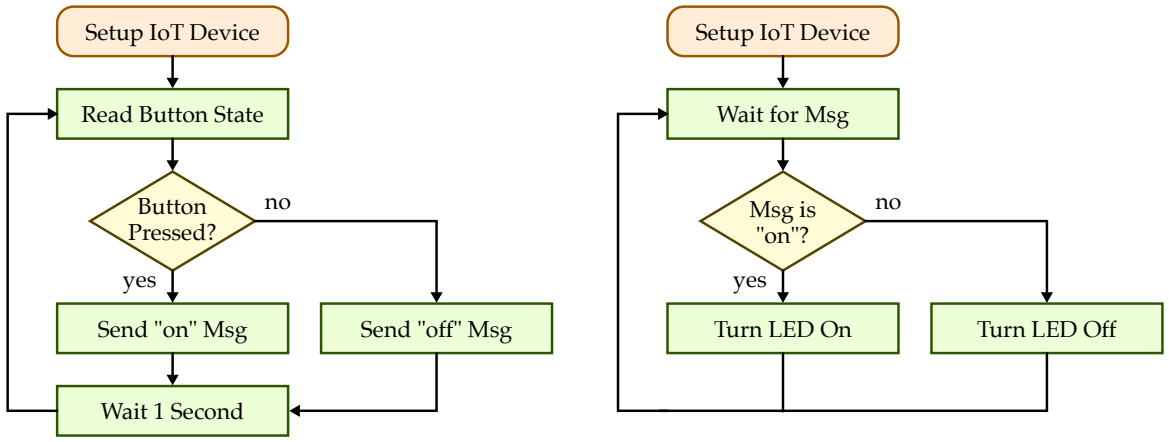
The Particle Argon has only two buttons. The reset button will restart the Particle Argon, and the mode button is only used for initial setup and debugging. In general, you will not need to use these buttons. Finally, there is an integrated blue LED wired to pin D7 which can be used for debugging. We have plugged the Particle Argon into a shield which makes it easier to connect input/output modules. The shield has a total of eight ports including two digital ports, three analog ports, two I2C ports, and one UART port. For our "smart light" system, we can plug the button input module into digital port D4 on the Particle Argon input device, and we can plug the LED output module into digital port D4 on the Particle Argon output device.

2. Algorithm: Flowcharts

At the application level, we have specified the high-level goal of building a "smart light" system. We now need to refine this high-level description into an actual *algorithm*. An algorithm is a step-by-step procedure for implementing the desired application. There are many ways to represent algorithms, but for this lab we will use a specific representation known as a *flowchart* which uses a set of shapes and arrows to visually express the steps in an algorithm. Although conventions can vary, in this lab we will use: an orange rectangle to indicate the very first step of the algorithm; a green rectangle to indicate a processing step; and a yellow diamond to indicate a decision step.

Figure 5(a) illustrates the algorithm for the IoT input device using a flowchart. After setup, the IoT input device will read the button state. If the button is pressed, then the algorithm will move along the "yes" arrow, and if the button is not pressed, then the algorithm will move along the "no" arrow. Depending on the state of the button, the algorithm will either send an "on" message or an "off" message to the IoT cloud. After sending the message, the algorithm waits one second to avoid sending messages to fast. There are often limits to the number and/or the rate of messages IoT devices can send and/or receive to/from the IoT cloud.

Figure 5(b) illustrates the algorithm for the IoT output device also using a flowchart. After setup, the IoT input device waits for a message. Once a message is received, the algorithm checks the content



(a) Flowchart for IoT Input Device (b) Flowchart for IoT Output Device

Figure 5: Flowchart for "Smart Light" Algorithm

```
1 int a;       // declaration
2 a = 2;       // assignment
3 int b = 3;   // initialization
4
5 int c;
6 c = a + b;
```

(a)

```
1 // function to add two integers
2 int add( int a, int b )
3 {
4   int sum;
5   sum = a + b;
6   return sum;
7 }
```

(b)

```
1 int c;
2 c = add( 2, 3 );
```

(c)

Figure 6: Example C++ Code Snippets

of the message. If the message is "on", then the algorithm turns the light on. If the message is "off", then the algorithm turns the light off. Then the algorithm waits for another message.

3. Programming Language: C++

Now that we have refined our application into an algorithm, we can implement this algorithm as a program in a specific *programming language*. There are many programming languages each with different advantages and disadvantages, but for this lab we will be using the C++ programming language. Figure 6 shows three example C++ code snippets. A program consists of a sequence of statements; these statements are executed one at a time by the computer to ultimately execute the program. Each statement is like a sentence in the English language; we read each sentence one at a time to understand a paragraph. In C++, a statement ends with a semicolon (`;`), while in the English language, a sentence ends with a period.

Figure 6(a) illustrates the most basic statements in the C++ programming language. The statement on line 1 is a *variable declaration* statement. We can think of a variable as a named "box" where we can store values. So in this example, we have a "box" named `a`. Each variable also has a *type* that indicates what kind of values can be stored in the box. In this case, the variable `sum` can only store values of type `int` which means the variable can store integer values (i.e., whole numbers). The statement on line 2 is a *variable assignment* statement. The value of the expression on the right-hand side of the equals operator (=) is stored in the variable on the left-hand side of the equals operator.

```
1  int button_state;
2  button_state = read_button_state( button_pin );
3
4  if ( button_state == 1 ) {
5    // send "on" msg
6  }
7  else {
8    // send "off" msg
9  }
10
11 // wait 1 second
```

(a) Sketch of IoT Input Device Program

```
1  void receive_msg( msg )
2  {
3    if ( msg == "on" ) {
4      // turn light on
5    }
6    else {
7      // turn light off
8    }
9  }
```

(b) Sketch of IoT Output Device Program

Figure 7: Sketch of C++ Programs for Smart Light

In this case, we store the value 2 into the variable `a` (i.e., we put the value 2 into the box named `a`). The statement on line 3 is a `variable initialization` statement which simply does a variable declaration and assignment in a single step. Once the computer has finished executing lines 1–3, the value 2 is stored in the variable `a` and the value 3 is stored in the variable `b`. Note that *comments* start with two forward slashes (i.e., `//`) and are ignored by the computer. You can feel free to exclude them although comments are important part of effective software engineering. Line 5 is another variable declaration statement. Line 6 is a variable assignment statement where the right-hand side is more complicated than just a single number. Here the computer must first determine the value stored in the variable `a`, the value stored in the variable `b`, add these two values together, and store the result (i.e., the value 5) in the variable `c`.

Figure 6(b) illustrates how to define a *function*. A function is a parameterized sequence of statements. Line 2 declares the function's interface including the function's name (`add`), the functions *parameters* (`a`,`b`), the type of each parameter (`int`), and the function's return type (`int`). Lines 3–7 are called the *function body*. In this example, the function first adds the values stored in the two parameters and stores the result in a variable named `sum`. The sum is then returned using a *return statement* on Line 6. Figure 6(c) illustrates how to call a function. We use the function's name and include a list of variables which will be passed in as the parameters to the function. So the example in Figure 6(c) will first declare a variable named `c`, then call the function `add`, which adds the values 2 and 3 before returning the result 5, which is finally stored in the variable `c`.

Figure 7 is a sketch of an initial C++ program for both the IoT input device and the IoT output device which corresponds to the algorithms shown in Figure 5. Lines 4–9 are a *conditional statement*. First, the conditional expression within the parenthesis on line 4 is evaluated (i.e., does the variable `button_state` contain the value 1?). If this conditional expression is true, then the statement on line 5 is executed. If this conditional expression is false, then the statement on line 8 is executed. A conditional statement can be used to implement decision steps in our flowchart. However, the C++ program in Figure 7 is not complete; it is just a sketch, since how our program reads the button state, sends messages, receives messages, and turns on/off LEDs depends on the operating system which is discussed in the next section.

If you would like to learn more about basic C++ programming consider reviewing this gentle introduction which is part of an intermediate programming course taught at Cornell University:

- `https://cornell-ece2400.github.io/ece2400-docs/ece2400-T01-intro-c`

4. Operating System: Particle OS

Figure 7 was an incomplete sketch of the programs needed to implement our flowchart algorithms. To complete the sketch, we will need to call various functions that are part of the *operating system*. The operating system is in charge of managing the interaction between the application program and the underlying low-level hardware. The Particle operating system (OS) provides functions for setting up the Particle Argon, reading/writing pins, and sending/receiving messages (also called events). We will start by considering a simple example that adds two numbers, before revisiting the complete C++ programs to implement our flowchart algorithms.

Figure 8 shows a C++ program which adds two numbers together and displays the result using an LED. The Particle Argon devices we are targeting do not have any kind of screen, so we have to think creatively about how to display the status of the program. Here we are blinking an LED `z` times where `z` is the sum of the variables `x` and `y`.

Our Particle Argon programs can be divided into four sections. The first section (lines 1–7) is where we create global names for pin assignments and define global variables (i.e., variables that can be accessed in any function throughout the program). The second section (lines 9–16) is where we define any helper functions (e.g., an `add` helper function to add two integers together). The third section (lines 18–24) is the `setup` function. The statements in the `setup` function execute only once at the very beginning of the program. The fourth and final section (lines 26–43) is the `loop` function. The statements in the `loop` function execute repeatedly over and over again. Throughout this lab, we will provide you the necessary code for the first three sections; you will be responsible for focusing on the `loop` function and possibly adding global variables where necessary. The Particle OS takes care of calling the `setup` and `loop` functions appropriately.

The statement on line 23 is a call to the `pinMode` function which is provided by the Particle OS. The `pinMode` function configures the given pin number to be either an input pin or an output pin. Since we assigned the value `D7` to the variable `led_pin`, we are effectively configuring digital pin `D7` to be an output. Digital pin `D7` corresponds to the small blue LED by the USB connector on the Particle Argon (see Figure 4). The iteration statement on line 34 is a for loop which is used to execute the loop body multiple times. This specific code will execute the loop body on lines 35–38 `z` times. The loop body contains two calls to the `digitalWrite` function and two calls to the `delay` function. Both of these functions are provided by the Particle OS. The `digitalWrite` function will write the corresponding output pin with either a `HIGH` or `LOW` voltage, while the `delay` function tells the Particle Argon to wait for the given number of milliseconds before continuing to the next statement. Finally, on line 42 we wait an additional four seconds.

Figure 9 shows the complete C++ programs that implement the algorithms shown in Figure 5. Notice how the digital pin `D4` is setup to be an input pin in the IoT input device program, but it is setup to be an output pin in the IoT output device program. The Particle OS `digitalRead` function is used to read the button state in the IoT input device program, and the Particle OS `digitalWrite` function is used to write the LED in the IoT output device program. The IoT input device program in Figure 9(a) uses the `Particle.publish` function to send messages (also called events) to the IoT cloud (see line 13 and 16). The IoT output device program Figure 9(b) uses the `Particle.subscribe` function to indicate what other function should be called to receive messages from the IoT cloud. Line 16 specifies that the `receive_msg` function should be called which is defined on lines 3–11. The Particle OS will call this function with the name of the event and the message as C strings. C strings can be quite tricky to work with, so care is required. On line 5, we use the `strcmp` function to compare the string variable `msg` to string literal `"on"`. The `strcmp` function returns 0 if the two strings match.

```
// Global constants for pin assignments and global variables

int led_pin = D7;

int x = 2;
int y = 3;
int z = 0;

// Helper functions

int add( int a, int b )
{
  int sum;
  sum = a + b;
  return sum;
}

// The setup routine runs once when you press reset

void setup()
{
  // Configure led_pin as digital output
  pinMode( led_pin, OUTPUT );
}

// The loop routine runs over and over again

void loop()
{
  // Do the addition
  z = add( x, y );

  // Blink LED z times
  for ( int i = 0; i < z; i++ ) {
    digitalWrite( led_pin, HIGH );  // Turn on the LED
    delay(500);                     // Wait 0.5 seconds
    digitalWrite( led_pin, LOW );   // Turn off the LED
    delay(500);                     // Wait 0.5 seconds
  }

  // Wait four seconds
  delay(4000);
}
```

Figure 8: Complete Example C++ Program

```
int button_pin   = D4;
int button_state = -1;

void setup()
{
  pinMode( button_pin, INPUT );
}

void loop()
{
  button_state = digitalRead( button_pin );
  if ( button_state == 1 ) {
    Particle.publish( "button_state", "on" );
  }
  else {
    Particle.publish( "button_state", "off" );
  }
  delay(1000);
}
```

(a) IoT Input Device Program

```
int led_pin = D4;

void receive_msg( const char* event, const char* msg )
{
  if ( strcmp( msg, "on" ) == 0 ) {
    digitalWrite( led_pin, HIGH );
  }
  else {
    digitalWrite( led_pin, LOW );
  }
}

void setup()
{
  pinMode( led_pin, OUTPUT );
  Particle.subscribe( "button_state", receive_msg );
}

void loop()
{
  // empty
}
```

(b) IoT Output Device Program

Figure 9: Complete C++ Programs for Smart Light

5. Compiler: Particle Development Environment

Now that we have refined our algorithm into a programming language with operating system support, we can use a *compiler* to translate the high-level program statements into the low-level instructions that the machine can actually execute. In this lab, we will be using the online Particle development environment to compile our programs. You can start the Particle development environment by going to this URL:

- `https://build.particle.io`

You should not create a new account. You should login with your group username and password. Figure 10 labels the key icons on the left-hand side of the Particle development environment:

- **Flash:** Compiles and flashes the current code to the selected device
- **Verify:** Compiles without flashing the current code
- **Save:** Saves the current code
- **Code:** Shows a list of available programs
- **Library:** Explore libraries
- **Help:** Does not work for our Particle Argon
- **Docs:** Brings you to the Particle documentation site
- **Devices:** Shows a list of all devices
- **Console:** Brings you to the Particle console for monitoring the IoT cloud
- **Settings:** Log out

The most important icon is the *flash* (lightning) icon in the upper-left hand corner. Clicking this icon will cause the Particle IDE to compile your program into machine instructions and then to upload the resulting machine instructions to the Particle Argon for execution. Figure 10 illustrates what happens

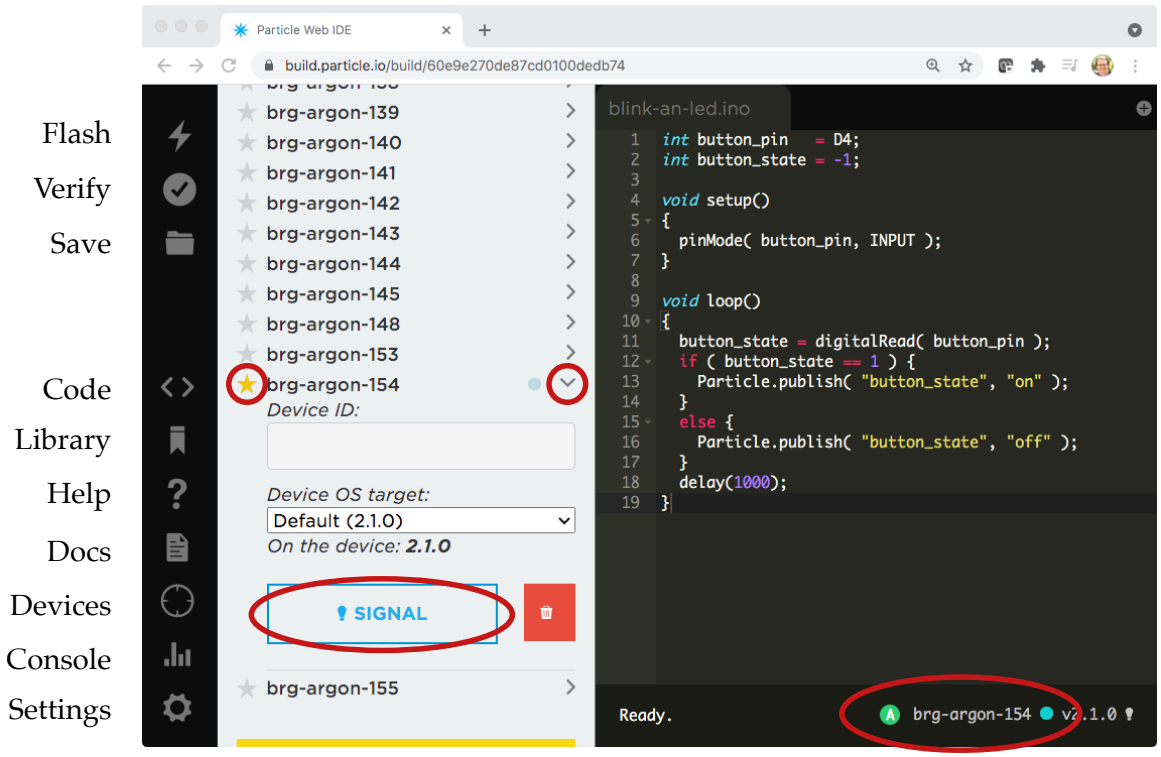


Figure 10: Particle Development Environment

when you click on the *devices* icon. Note that you *must* select your specific device by clicking the little star next the device that matches your device number. You can always look at the bottom right-hand corner to verify that you are working on your specific device. You can click the little > to reveal the *signal* button. Clicking this button will cause your Particle Argon to blink rainbow colors which is a great way to confirm that your Particle Argon is connected to the internet and working.

Clicking the *console* icon will take you to the Particle Console which enables monitoring what is going on in the Particle cloud. To start, you probably want to first click on the devices icon in the upper-left hand corner of the Particle console and then choose your device. Figure 11 illustrates what the device page looks like in the Particle Console. In the upper-right hand corner you can click the *signal* button to signal your Particle Argon and case the status LED to blink rainbow colors. The *last vitals* pane is a useful way to verify that your Particle Argon device has a strong conenction to the WiFi network. You can monitor messages (also called events) being sent into the Particle cloud. You can click on an event to see more information. You can also update Particle variables in the lower-right hand corner. We will learn more about Particle variables in the actual lab session.

Take a few minutes to watch these two video tutorials:

- `https://www.youtube.com/watch?v=b6sUP16HWKM`
- `https://www.youtube.com/watch?v=QFXK7RvDT-Y`

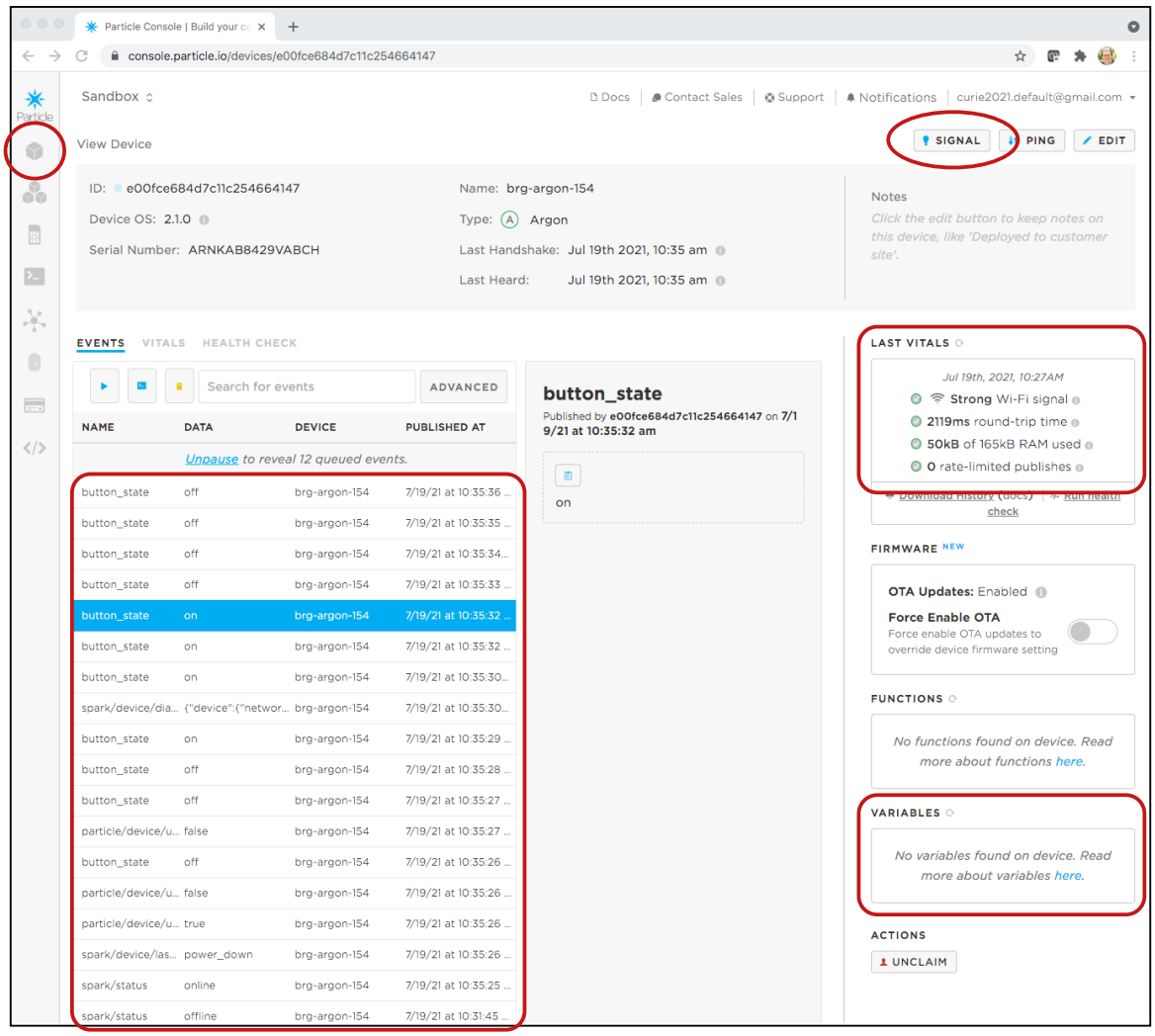


Figure 11: Particle Console Device Page

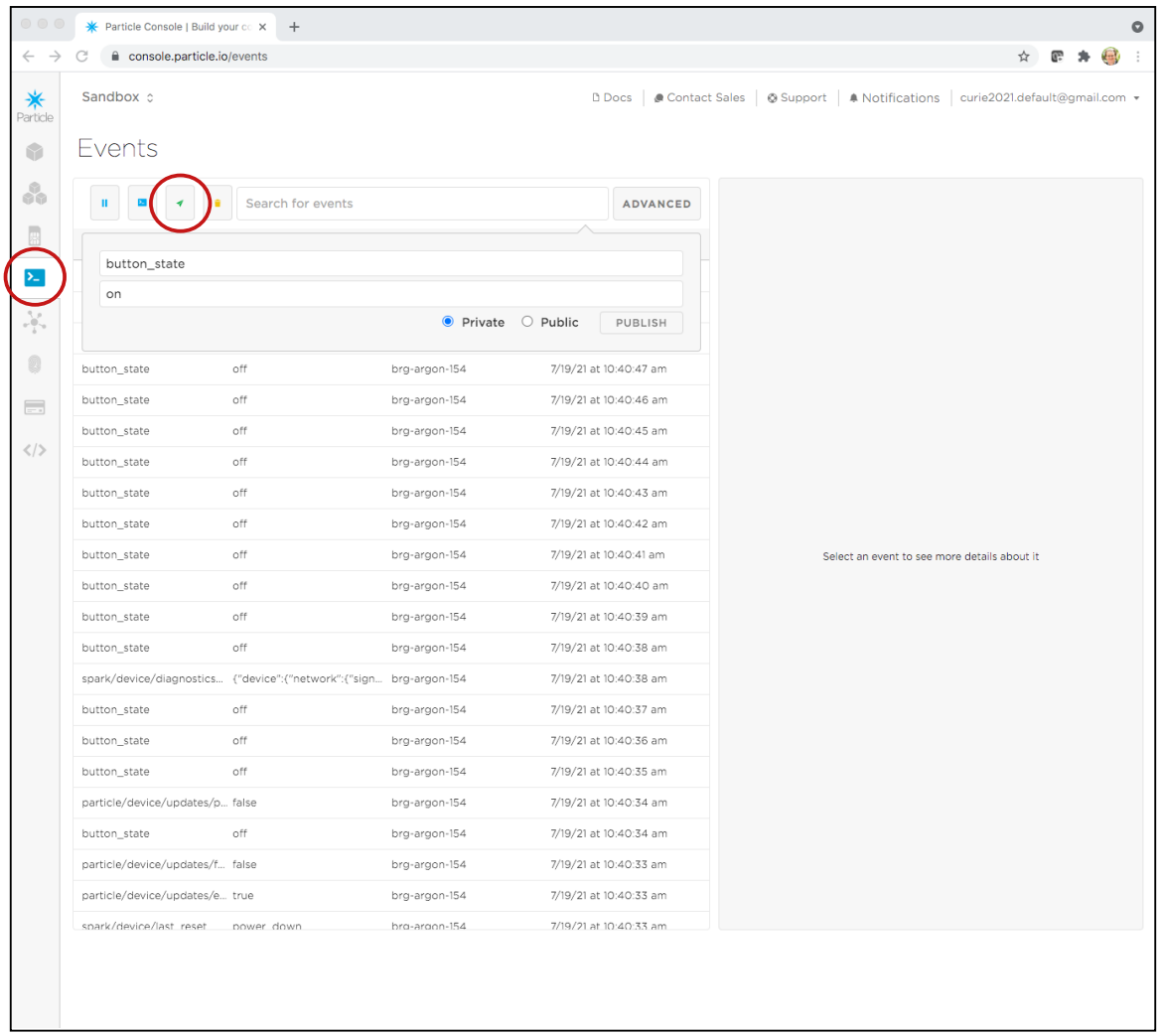


Figure 12: Particle Console Event Page

6. Instruction Set Architecture: ARM Machine Instructions

The compiler translates the high-level statements used in the programming language into very simple machine instructions which are part of an *instruction set architecture*. Figure 13 illustrates a very simple architecture with a processor to compute on data and two kinds of memory to store data: *main memory* is slow but can store a large amount of data, while *registers* are fast but can only store a small amount of data. This simple architecture supports three kinds of machine instructions: *load instructions* move values from memory into registers; *store instructions* move values from registers into memory; and *arithmetic instructions* perform simple arithmetic on values stored in registers.

As mentioned in the previous section, the compiler transforms the sequence of high-level statements in Figure 6 into a sequence of machine instructions the processor can understand. Figure 14 shows the actual machine instructions generated by the compiler for line 5 in Figure 6(b). There are four machine instructions: the first two machine instructions load values from main memory into registers, the third machine instruction adds these two values together, and the final machine instruction stores the sum back out to main memory. Note that the processor can use a ripple-carry adder similar to the one you developed in Lab 1 to implement the add machine instruction which means our tour of the computer systems stack is complete! In Lab 1, we explored technology to machine instructions, and in lab 2 we explored from applications to machine instructions. **We meet in the middle of the stack at the instruction set architecture, where hardware meets software and software meets hardware!**

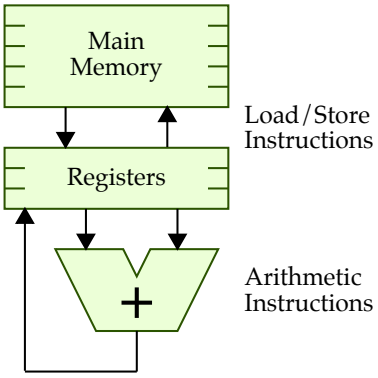


Figure 13: Simple Architecture

```
1 # load two values from main memory into two registers
2 ldr     r2, [r7, #4]
3 ldr     r3, [r7]
4
5 # do the actual addition
6 add     r3, r3, r2
7
8 # store the sum from a register back into main memory
9 str     r3, [r7, #12]
```

Figure 14: Machine Instructions for Line 5 in Figure 6(b)