

Widget Studio 1 - Introduction to Systems Prototyping Using Electronics and Computing

1 Introduction

Welcome to the first Widget Studio. We will be introducing the framework we will be using for Widget Studios while also helping you build some skills in systems prototyping using electronics and computing.

1.1 Skill Building and Challenges

Each Widget Studio will typically consist of a skill-building portion where we introduce concepts and practices skills with no particular context in mind. There will then be a challenge scenario where you solidify your understanding by applying what you have learned to a particular context. This may involve designing a more complex system than the example system we use in the skill building portion. We will provide hints and try to explain any new concepts needed for the challenge scenario.

1.2 Framework for Approaching Prototyping

Throughout the document, we will frame systems prototyping from two perspectives

1. What system goals are and what the system must be and do to achieve its goals. These focus on system functions/performance, features, and how it interacts with its operational environment.
2. What the engineer must do to develop a system that meets those goals. These are specific tasks to perform as well as concepts, tools, materials, reference information the engineer must leverage in the design and implementation of the system.

1.3 Terminology and Document Structure

Throughout the document, we will use specific terminology as we structure the approach to systems prototyping. The main ones are:

1. **Tools:** These are the pieces of equipment or software you will need to put your system together and to test it.
2. **Components and Materials:** These refer to the physical components or materials and also software that will eventually become part of your implemented system.
3. **Reference Materials:** These are the documents you would need to consult to find useful information about your tools, components, or materials.
4. **Tasks:** These are the actions you need to take in order to design, implement, and test your system.

We will also try to highlight places where

1. You should pause to think/reflect on an action you just took or a concept you just reviewed
2. You should record information that will be useful later in the design.
3. You should answer some questions to help you solidify your understanding of concepts or tasks.

Contents

1	Introduction	1
1.1	Skill Building and Challenges	1
1.2	Framework for Approaching Prototyping.....	1
1.3	Terminology and Document Structure	1
2	Explore Abstract System and Implementation Considerations.....	5
2.1	Brief Systems Theory Primer	5
2.2	Example System for the Studio	6
2.2.1	Abstract representation	6
2.2.2	Implementation Considerations.....	6
3	Prepare Tools and Materials.....	7
3.1	Tools.....	8
3.1.1	Python 3	8
3.1.2	Mu Editor.....	8
3.2	Materials and Components.....	8
3.2.1	Raspberry Pi Pico.....	8
3.2.2	CircuitPython	9
3.2.3	Breadboard	9
3.3	Tasks	9
3.3.1	Download and install Python 3.9.7	9
3.3.2	Download and install Mu Editor (version 1.1.0-beta-5).....	10
3.3.3	Review Notes on Electrical Safety	11
3.3.4	Download and install Circuit Python (version 7.0.0-rc.1).....	13
3.3.5	Verify Installation of CircuitPython (LED Blink).....	14
4	Implement and Test System Output	16
4.1	Tools.....	16
4.2	Components and Materials.....	16
4.3	Reference Material	17
4.4	Tasks	17
4.4.1	Review GPIO concepts	17
4.4.2	Design the circuit implementation to be compatible with the materials.	17
4.4.3	Implement the designed circuit on the breadboard.....	21
5	Implement and Test System Input.....	22
5.1	Tools.....	22

5.2	Components and Materials.....	22
5.3	Reference Material	22
5.4	Tasks	23
5.4.1	Review active low and pull-up resistor concepts	23
5.4.2	Build button input circuit, update code, and test	24
6	Challenge Scenario.....	26
6.1	Background	26
6.2	Task	26
6.2.1	Consider the abstract features of the system	26
6.2.2	Consider the System Implementation	28
6.3	Deliverables.....	29
7	References	29

2 Explore Abstract System and Implementation Considerations

We will explore some basic systems theory concepts and connect them to the work we will be doing the rest of the studio. The only tools required are pencil/pen and paper or equivalent materials to write and follow along.

2.1 Brief Systems Theory Primer

In systems theory, loosely^{1,2}, a system is an entity that

1. Takes in an input (usually a signal typically denoted as $u(t)$)
2. May have internal state (usually denoted as $x(t)$)
3. An output it produces (usually a signal typically denoted as $y(t)$)
4. A function/behavior (typically denoted $F(x(t), u(t))$) that dictates how the system responds to input by changing state or producing outputs or both.

A system may be considered as an atomic unit or a combination of smaller systems. A system may have an architecture, which shows how its various parts are organized. Typically, we visually represent a system using block diagram as shown below.

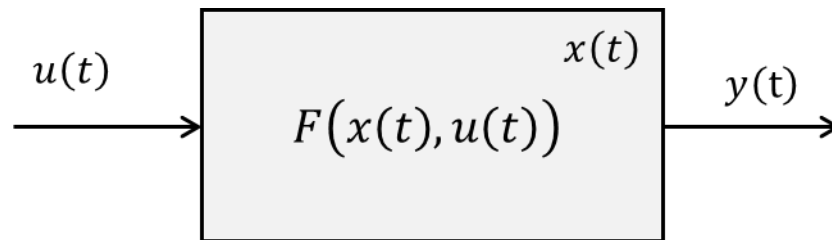


Figure 1. Block diagram representation of system.

Implied implementations of systems can be represented by more domain specific diagrams such as electrical schematics (as we will do in this studio), mechanical diagrams, blueprints, etc.

¹ For an intuitive introduction to systems concepts, see this reference (still under construction): Philip Asare, "Systems" in [Essence and Accidents of Engineering Embedded and Cyber-Physical Systems](https://eg.bucknell.edu/~encpsresource/ena/systems.html). URL: <https://eg.bucknell.edu/~encpsresource/ena/systems.html>

² For more formal treatment of systems concepts and representations of systems, their architectures and behaviors, see the following references:

Edward A. Lee and Pravin Varaiya, [Structure and Interpretation of Signals and Systems](https://www.eecs.mit.edu/publications/1999/structure-and-interpretation-of-signals-and-systems/), Second Edition, LeeVaraiya.org, ISBN 978-0-578-07719-2, 2011.

Edward A. Lee and Sanjit A. Seshia, [Introduction to Embedded Systems, A Cyber-Physical Systems Approach](https://mitpress.mit.edu/9780262533812/introduction-to-embedded-systems-a-cyber-physical-systems-approach/), Second Edition, MIT Press, ISBN 978-0-262-53381-2, 2017

The specification of what system expects on input or produces on an output is called an interface specification. To interact with the system properly you must obey the interface specification.

2.2 Example System for the Studio

The example system we will be exploring in the studio is one that turns on an LED when a button is pushed down and turns it off when the button is released.

2.2.1 Abstract representation

We can represent the input, $u(t)$, as a continuous signal that takes on one of two values at any point in time (i.e. “pressed” or “not pressed”) and the output, $y(t)$, as a continuous signal that also takes on one of two values at any point in time (“on” or “off”). The function $F(u(t))$ can be represented using the table below. Notice that this system has no state since all it does is change the output in response to the input immediately (i.e. the current output, $y(t)$, only depends on the current input, $u(t)$).

Table 1. Function $y(t) = F(u(t))$ of example system

$u(t)$	$y(t)$
pressed	on
not pressed	off

We can also represent the system response over time using the following timeline diagram.

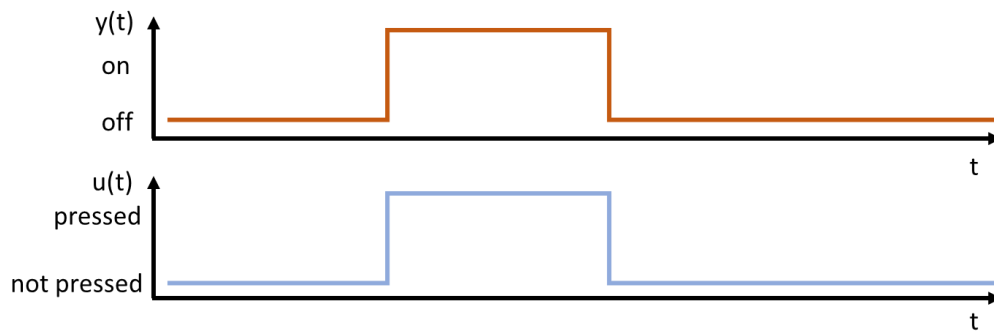


Figure 2. Timeline diagram representation of system function $y(t) = F(u(t))$

2.2.2 Implementation Considerations

There are a number of different ways to implement the abstract system described above. The implementation we will explore in the studio is one that uses electronics and computing as shown below.

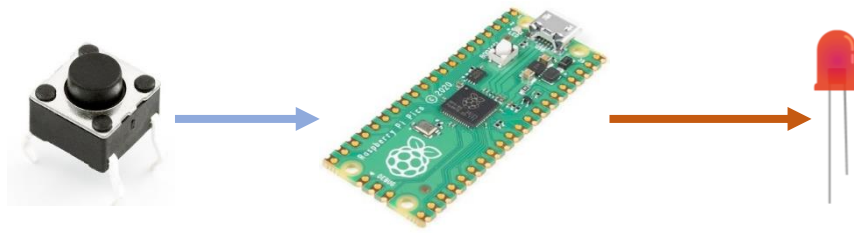


Figure 3. System concept diagram

The button implements the input, and the LED implements the output. The computer (called a microcontroller (MCU)) implements the function. The software directs the microcontroller to continuously check the state of the button input (by checking the voltage on the pin the button is connected to) and to turn the LED on or off (by controlling the voltage on the pin that the LED is connected to).

Below are two other representations of the same system. Each shows more details about the implementation.

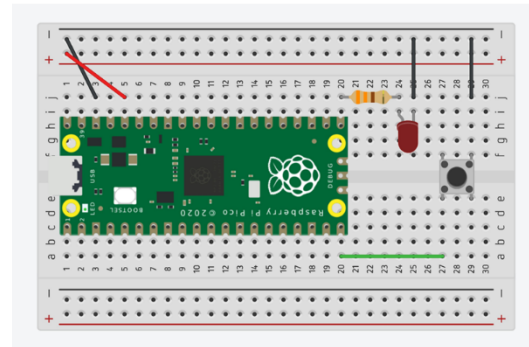
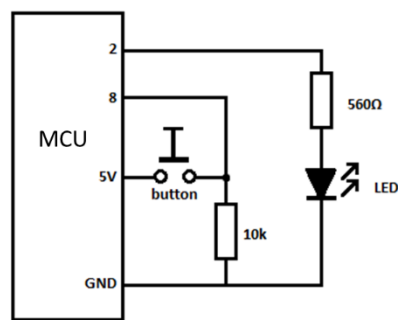


Figure 4. Other representations of system architecture. Left: circuit schematic. Right: breadboard implementation



Here are some things to think about:

1. How would you implement this using electronics but without the microcontroller?
2. How would you implement a similar kind of system (one that has two potential input and two potential output values, and where continuously applying one input value continuously produces a corresponding output value) using only mechanical components?

3 Prepare Tools and Materials

Before we dive into implementing our system, it is important to make sure we have the right tools set up properly and have the required components and materials.

3.1 Tools

3.1.1 Python 3

Many of the other software tools we will use depend on the Python 3 ecosystem. You may also use Python 3 in later studios to help with data analysis and other software tasks.

3.1.2 Mu Editor

Mu is the integrated development environment (IDE) that we will use to develop some of the software that goes into our systems. Mu Editor is recommended by the developers of CircuitPython for editing and working with CircuitPython code (more on CircuitPython below in the [Materials and Components section](#)). Technically, you can use any text editor or Python IDE to edit your code, but there have been problems reported with some depending on the way they write out saved files to the Pi Pico's flash storage.

3.2 Materials and Components

3.2.1 Raspberry Pi Pico

The Raspberry Pi Pico is a small, inexpensive microcontroller board. As part of Praxis III, you'll learn to use two microcontroller boards, both based on the Raspberry Pi RP2040 microcontroller. For the first half of the course, you'll primarily be using the Raspberry Pi Pico as you work through a set of studio activities designed to help you gain experience programming and doing electronics and mechanical design. We categorized it as a component because it will become part of the deployed system.

Technically, the microcontroller is just the "brain" chip at the center of the microcontroller board, which includes power management circuitry, connectors, USB controller chips, memory, general purpose input/output (GPIO) pins, and other peripherals that allows the microcontroller to interact with the outside world easily and conveniently. You may sometimes hear the entire board referred to as the microcontroller. Each student will get a Raspberry Pi Pico as part of this lab. You can read more about the Raspberry Pi Pico, the RP2040 Microcontroller on the [Raspberry Pi Pico website](#).



Figure 5. Picture of the Raspberry Pi Pico development board

3.2.2 CircuitPython

CircuitPython is a version of the Python programming language that works on microcontrollers such as the Raspberry Pi Pico. It allows us to reuse much of what we know about programming in Python in the microcontroller programming environment. We will highlight some of the differences as we go along.

3.2.3 Breadboard

To use the microcontrollers with other electromechanical components, you'll often use a breadboard. A breadboard is a small plastic shell with holes in the top and conductive strips running through the interior. Typically, the holes have a 0.1" spacing. Breadboards are commonly used for prototyping electrical circuits, as microcontrollers, integrated circuits, and passive components can be easily attached and detached. An image of a breadboard is shown in Figure 6 with back removed to show the location of the conductive strips. The arrows point to the power and ground rails on the outside of the board, which are used to make wiring easier/more organized when multiple components need to be connected. The interior strips allow components to be connected to each other easily without having to solder or twist wires together.

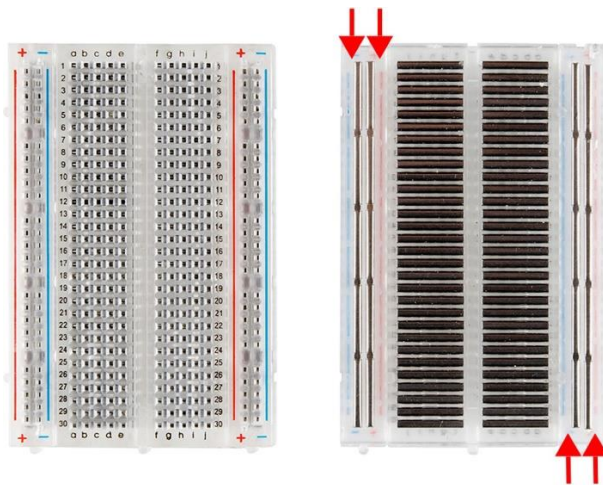


Figure 6. Breadboard top (left) and bottom with backing removed (right).
Contiguous dark lines on the right diagram indicate electrical connections.

3.3 **Tasks**

3.3.1 Download and install Python 3.9.7

The version of Python we will be using through the term is Python 3.9.7. **It is important that everyone uses the same version** so we can help troubleshoot problems and do not run into compatibility issues.

Windows and Mac OS

Installation is relatively simple on Windows 10 and Mac OS, just download the installation file for your respective OS from <https://www.python.org/downloads/>. On Windows, when the option to disable the PATH length limit comes up, you should do so.

Files					
Version	Operating System	Description	MDS Sum	File Size	GPG
Gzipped source tarball	Source release		798b9d3e866e1906f6e32203c4c560fa	25640094	SIG
XZ compressed source tarball	Source release		ecc29a7688f86e550d29dba2ee66cf80	19051972	SIG
macOS 64-bit Intel installer	macOS	for macOS 10.9 and later	d714923985e0303b9e9b037e5f7af815	29950653	SIG
macOS 64-bit universal2 installer	macOS	for macOS 10.9 and later, including macOS 11 Big Sur on Apple Silicon (experimental)	93a29856f5863d1b9c1a45c8823e034d	38033506	SIG
Windows embeddable package (32-bit)	Windows		5b9693f74979e86a9d463cf73bf0c2ab	7599619	SIG
Windows embeddable package (64-bit)	Windows		89980d3e54160c10554b01f2b9f0a03b	8448277	SIG
Windows help file	Windows		91482c82390caa62accfdacbaabf618	6501645	SIG
Windows installer (32-bit)	Windows		90987973d91d4e2cddb86c4e0a54ba7e	24931328	SIG
Windows installer (64-bit)	Windows	Recommended	ac25cf79f710bf31601ed067ccd07deb	26037888	SIG

Figure 7. Screenshot of download page of Python website.

Linux

On Ubuntu, it is easier to install Python using a package manager such as Apt. The guide here: <https://linuxize.com/post/how-to-install-python-3-9-on-ubuntu-20-04/> gives instructions on how to do so. If you are using a different distribution, you should be able to follow similar steps using the package manager of your choice.



Again, make sure that you are downloading and installing Python 3.9.7

3.3.2 Download and install Mu Editor (version 1.1.0-beta-5)

You can download and install Mu Editor with the packages & instructions found [here](#). The version we are using is Mu-1.1.0-beta-5.

The first time you open Mu Editor, it may take some time to run. Once installation is complete, you are ready to plug in your Raspberry Pi Pico.

Windows Installation Installing on Windows, you may get one of the two following messages:

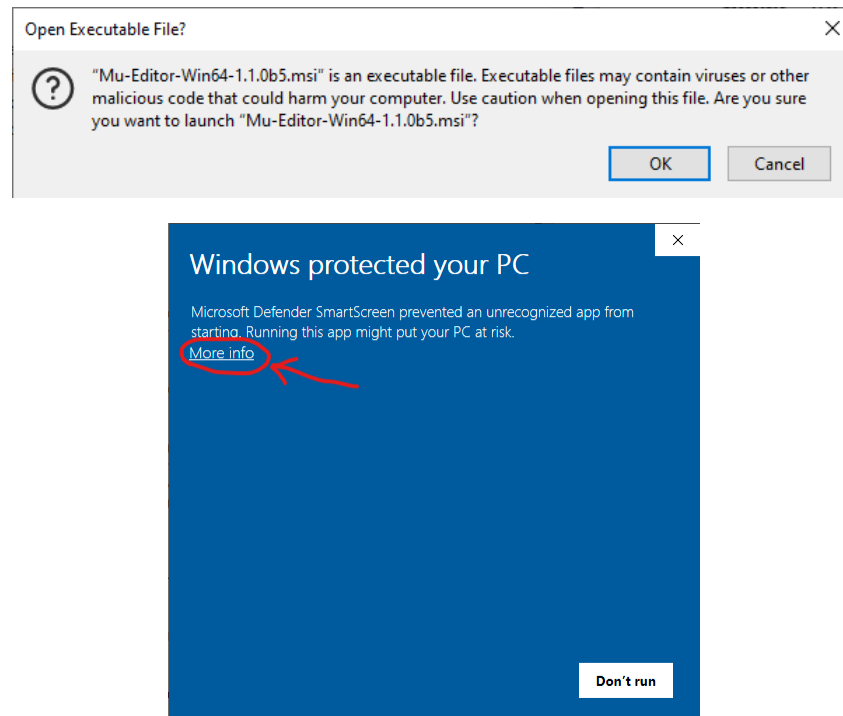


Figure 8. Common system protection messages when installing on Windows.

If you do, click OK on the first one to continue. On the second, click **More info**, then the **Run Anyway** button to install. Then, accept the license agreement and continue the installation process.



NOTE: Some people have reported issues downloading Mu Editor with Microsoft Edge. If you run into issues, try using another browser such as Firefox or Chrome.



Again, make sure that you are downloading and installing Mu-1.1.0-beta-5

3.3.3 Review Notes on Electrical Safety

Since this should be the first time you're handling your microcontroller, it's important to go through a few things with regards to electrical safety so that you don't inadvertently damage your components. Most microcontrollers and semiconductor devices, while used for a wide range of tasks in different environments, are quite sensitive to electrical damage. Overvoltage, overcurrent, electrostatic discharge, and thermal failure are some of the more common ways these devices can be damaged or fail completely. The microcontrollers used in this course are similarly vulnerable. This section includes some basic information on the four failure common failure modes that were just mentioned, and what you can do to prevent them.

Overvoltage

Overvoltage is when a component is exposed to a higher voltage than it is rated to handle. This will often occur due to an incorrectly configured power supply setup or improper grounding. Higher than expected voltages will cause the current paths and gates in the semiconductor to be overwhelmed and break down, causing electricity to flow in unexpected ways within the integrated circuit.

Overcurrent

Overcurrent happens when more current than is expected flows through a circuit. This can commonly be caused by excessive loading or short circuits. Most microcontrollers will have a limited ability to supply current to connected devices and handle current coming from external devices. **The RP2040 microcontroller has a limitation of 50 mA total across all pins [1].**

Short circuits commonly occur when pins are accidentally (or purposefully) connected without enough resistance between them to limit current. Remember Ohm's Law, $V=IR$. If the voltage is positive and the resistance is 0, the current will be infinite!

Electrostatic Discharge

Electrostatic Discharge (or ESD) occurs when a built-up charge is discharged through a component. Have you ever zapped yourself with static electricity after wearing socks on a carpeted floor? This is the same mechanism that leads to ESD. These shocks often hurt because you're discharging several hundred or thousand Volts, but with such a small current over such a short time that they can't significantly harm a human. A semiconductor-based microcontroller, however, can easily be damaged by these voltages. Even lower voltage discharges that you may not necessarily feel can be enough to damage components.

Thermal Failure

When power is dissipated in a component, a portion of it is dissipated as heat. If components overheat, electrical connections can degrade or melt completely, leading to failure.

Electrical Safety Best Practices

Below is a (non-exhaustive) list of some best practices to try to follow when working with microcontrollers and low-power electric circuitry.

1. Whenever possible, confirm that the voltage being used in a circuit is what you expect by using a multimeter to check it.
2. Always confirm that all ground points in circuits are electrically connected.
3. Make sure to use appropriate resistors for limiting current to and from microcontroller pins. If pull-up or pull-down resistors are needed on inputs, make sure they're configured properly in your code so that input pins aren't left "floating".

4. Always confirm sensors and connected devices interface with the same logic levels (voltages) that your microcontroller does. Use voltage dividers if necessary to ensure compatibility.
5. Avoid touching a microcontroller with anything conductive while it's operating to avoid accidentally shorting pins together.
6. Always place the microcontroller on a non-conductive surface when in use. There are exposed connections on the bottom of the board that can be shorted.
7. Don't leave unconnected/unjacketed wires dangling from breadboards, disconnect them completely or cover them with an insulator such as electrical tape.
8. If using more than a few components, power them using a separate power supply and control devices with relays or transistor-based switches.
9. Before picking up a microcontroller or touching circuitry, discharge yourself by touching a grounded piece of metal, such as the case of your computer or a metal table leg.
10. Check component datasheets for thermal limitations when available. Use heat spreaders or radiators when needed on high-power components.


3.3.4 Download and install Circuit Python (version 7.0.0-rc.1)

We will be using the 7.0.0-rc.1, which is a pre-release version. We must use this version because it supports features that we need for later studios that the more stable release does not support, *at the risk* that we run into some bugs because it has not been tested as extensively as the stable release. You can find out more about this release [here](#).

The instructions in this section will guide you through installing the CircuitPython firmware on your Raspberry Pi Pico.

1. Download the most recent beta release of the Raspberry Pi Pico CircuitPython firmware (7.0.0-rc.1 at the time of writing) from [this site](#), and copy it to the drive (the transfer may be a little slow, this is normal since the Pico uses USB1.1). Reboot the Pico by power cycling it once the transfer is complete.
2. Using a microUSB cable, plug one end into the Raspberry Pi Pico.
3. Press and hold the BOOTSEL button on the Pico while plugging it into your computer. It should show up as a USB drive device named "RP2 Boot" or something similar. There should be two files in the drive, *INDEX.HTM* and *INFO UF2.TXT*.
4. If the transfer was successful, the drive name should change to CIRCUITPY and you should see the files shown in Figure 9. If you only see a few of them, it's okay, as on some OS's some of the files and folders may be hidden by default. If you see the *lib* folder, *boot out.txt* file, and *code.py* file, you should be good to go!

-

 **Again, make sure that you are downloading and installing CircuitPython version 7.0.0-rc.1**

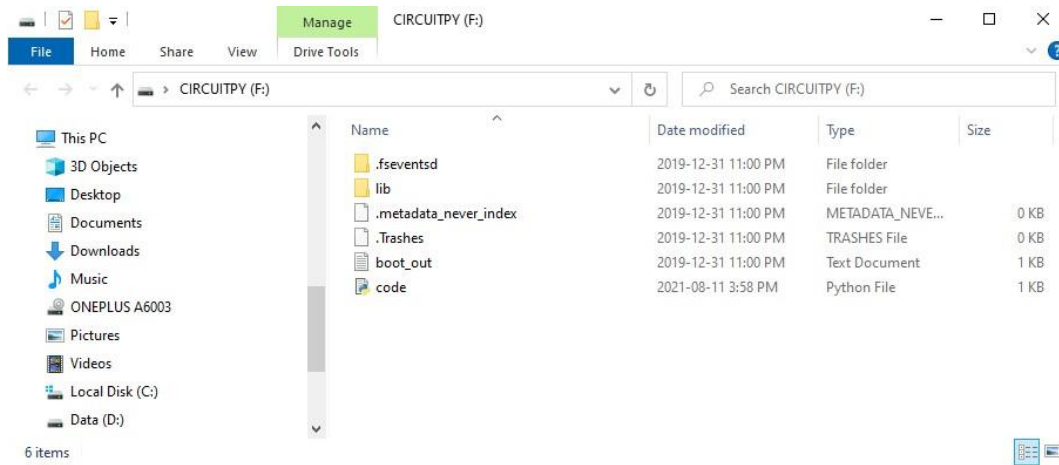



Figure 9. Files on the Raspberry Pi Pico after CircuitPython firmware installation.

3.3.5 Verify Installation of CircuitPython (LED Blink)

We will verify that CircuitPython firmware installed properly by writing code for our Pico. This also allows us to test if the Pico is working properly. For this we'll use Mu Editor to edit the `code.py` file on the Pico and blink the built-in green LED on the board. The CircuitPython firmware we installed in the previous step is configured to automatically run the `code.py` file at bootup. For any code you write to be executed by the Pico, you'll need it to either be in `code.py` or in another file whose code is called by `code.py`.

 Since we aren't using any external components, **you technically don't need a breadboard for this step, but we recommend you use one** to prevent yourself from accidentally touching the board's headers and shorting pins.

1. With the Pico plugged in via the USB cable, open the Mu Editor and select RP2040 mode.

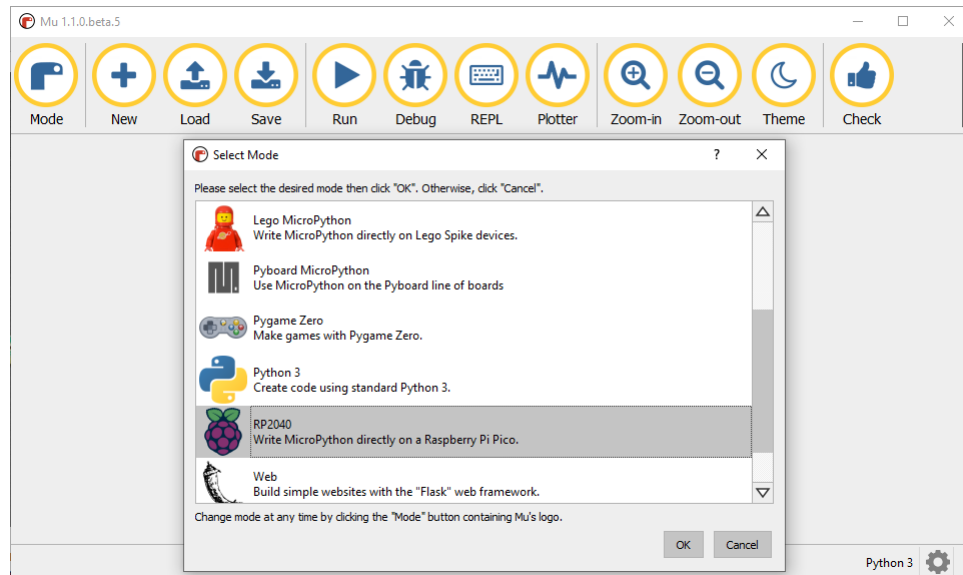


Figure 10. The Mu Editor startup window

2. Click the **Load** button on the top action bar and navigate to the drive containing the Pico's memory. Open the file `code.py`
3. Delete all text in the file, and then copy the text below (if you can't copy it from this pdf easily, there is a text file on Quercus containing the code). Make sure to use format with proper indenting if you copy the text.
4. Click the **Save** button on the top action bar to save your code.
5. Click the **Run** button on the top action bar to run the code with the serial console at the bottom of the editor open. You should see the LED on the Pico blinking slowly, and the message you typed in the print statement should appear in the console window.

LED Blink Code Snippet

```
# Import libraries needed for blinking the LED

import board
import digitalio
import time

# Configures the internal GPIO connected to the LED as a digital output
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

# Print a message on the serial console
print('Hello there! My LED is blinking now.')

# Loop so the code runs continuously
while True:
    led.value = True    # Turn on the LED
    time.sleep(0.5)     # wait 0.5 seconds
    led.value = False   # Turn off the LED
    time.sleep(0.5)     # wait 0.5 seconds
```

4 Implement and Test System Output

Now that we have our tools, components, and materials ready to go we can move on to implementing our system. We will first focus on implementing and testing the output functionality. The goal is to use a GPIO pin on the Pico board to light up an external LED

4.1 Tools

You will need the following tools:

- Mu Editor
- Digital Multimeter

4.2 Components and Materials

You will need the following components and materials. Make sure to collect them from your studio TAs and make sure the quantities you have collected have been recorded properly.

- a) 1x Raspberry Pi Pico with soldered headers
- b) 1x MicroUSB cable
- c) 1x Breadboard (small or large)
- d) 1x LED
- e) Resistors (1x 220 Ω , 1x 330 Ω , 1x 1k Ω)
- f) Jumper Wires (1 red, 3 black, 1 any other color, 1 any other color)

4.3 *Reference Material*

You'll need the following reference material to help you make decisions about your implementation design (links hyperlinked):

- [RP2040 Datasheet](#)
- [LED Datasheet](#)

4.4 *Tasks*

4.4.1 Review GPIO concepts

The general-purpose input-output (GPIO) allows a microcontroller to interact with the outside world using two voltage levels. It can either produce a voltage output or read an input voltage. One voltage level is considered "high" (or logical 1 or True in software or sometimes called "on"), which is represented by the high logic level voltage. For the Pico, the nominal value is 3.3V but in reality any voltage greater than 2.62V will register as "high". The other voltage level is considered "low" (or logical 0 or False in software or sometimes called "off"), which is represented by the low logic level voltage. For the Pico, the nominal value is 0V but in reality any voltage less than 0.5V will register as "low".

GPIO allows for software control of external devices, as we will be doing with the LED, or for software to take input from an external device, as we will be doing with the button, later. It is called general-purpose because it can be used for a wide range of external devices (not just buttons and LEDs) and because we are using software, we can choose to interpret inputs in various ways or produce a variety of outputs.

4.4.2 Design the circuit implementation to be compatible with the materials.

Before we implement the circuit, we must make some design decisions. The schematic for the circuit we would like to implement is shown in Figure 11. The Pico has some specifications about what is possible on its GPIO pins in terms of voltage and current (we are using the pin as a power source). The LED also has some specifications on what will make it light up. We need to make sure that we are connecting a two in a way that works for both components so we don't damage them, but we are also able to achieve the function we want.

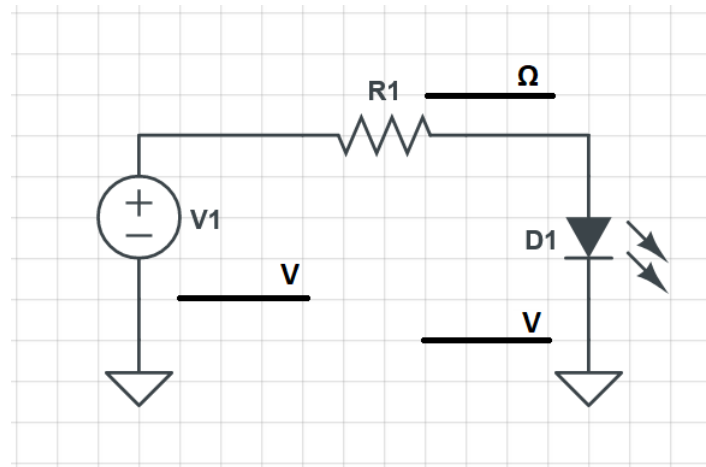


Figure 11. Schematic of circuit to implement

1. Examine the LED datasheet, and find voltage drop across it (sometimes called forward voltage) at various currents. Keep this information handy.



Think about how you could figure out the LED voltage drop at a specific current if you didn't have a datasheet.

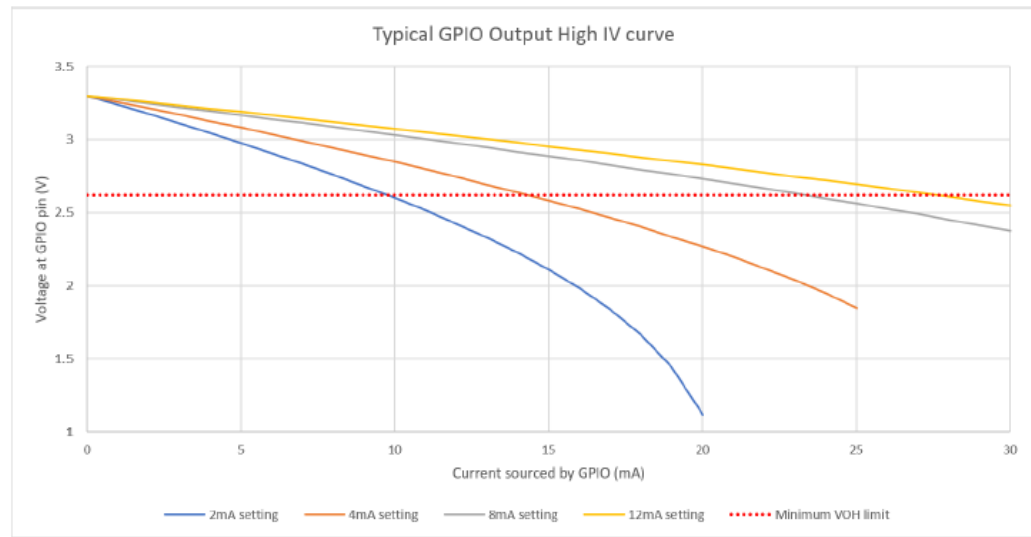
2. Determine the current you want the GPIO pin to output to *both light up the LED and prevent overcurrent* that will damage the microcontroller. There is actually a bit of confusion on what the maximum current a single pin on the RP2040 can provide before damaging the chip (see [discussion in this forum](#)), so we will try to sort this out.
 - a. The RP2040 datasheet (Section 5.2.3) provides some information on the GPIO pin specification.
 - b. Table 632 in Section 5.2.3.4 (at the bottom on page 634) says that the total current that drawn (sourced) across *all* pins should not exceed 50mA.

Maximum Total IOVDD current	I_{IOVDD_MAX}		50	mA	Sum of all current being sourced by GPIO and QSPI pins
-----------------------------	------------------	--	----	----	--

- c. Section 5.2.4.5 provides some notes on interpreting the GPIO specifications.

- i. It says that although the GPIO peripheral claims to have different current drive strengths (2mA, 4mA, 8mA, and 12mA), these really are not hard limits. Rather it provides a graph (Figure 169) of how the GPIO behaves at these drive strengths when you try to source different amounts of current.


Figure 169. Typical Current vs Voltage curves of a GPIO output.



- ii. The default drive strength (the value on Reset) is 4mA (according to Table 351 on Page 322 with the title “PADS_BANK0: GPIO0, GPIO1, ..., GPIO28, GPIO29 Registers”).

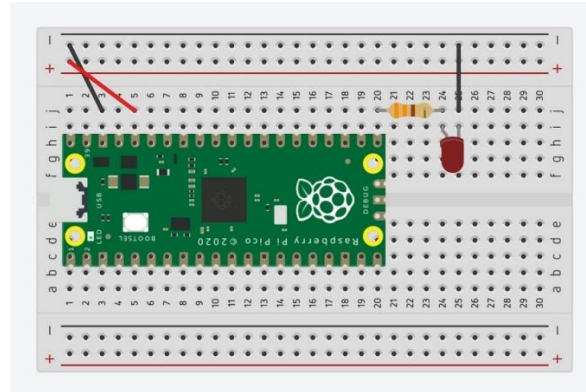
Table 351. GPIO0, GPIO1, ..., GPIO28, GPIO29 Registers

Bits	Name	Description	Type	Reset
31:8	Reserved.	-	-	-
7	OD	Output disable. Has priority over output enable from peripherals	RW	0x0
6	IE	Input enable	RW	0x1
5:4	DRIVE	Drive strength. 0x0 → 2mA 0x1 → 4mA 0x2 → 8mA 0x3 → 12mA	RW	0x1

- iii. The red line in the figure indicates the lowest voltage at which the software considers the GPIO pin value to be the high voltage (or logical 1 or True). **You want to pick a current that ensures the voltage is above this line.**
3. Look at the current you picked in step 2. Make sure this current still allows the LED to light up.
4. Given the information from Step 2 and 3, determine the value of the resistor to use to limit the current coming out of the GPIO pin. The value may require combining some of the resistors you got from the TA in series or parallel. You will need to determine the LED forward voltage based on the forward current from the datasheet. This [article from SparkFun](#) provides some information on how to determine the value of the resistor using the LED forward voltage. You may have to adjust your desired current as necessary to work with the resistors you have, and your final resistor value does not need to be exactly what you calculate; close enough is good enough.
5.  Based on the information above, make note of your expectations for the following (using what you know of circuit theory and information from previous steps to determine these values):
 - a. What do you expect the current flowing through the LED?
 - b. What should the voltage be
 - i. at the GPIO pin/before the resistor?
 - ii. after the resistor/before the LED?
 - iii. after the LED?

4.4.3 Implement the designed circuit on the breadboard.

1. Build the circuit as show below using the resistor value chosen (remember you may have to combine resistors in series or parallel).



2. Plug the Pico into your computer via the microUSB cable.
3. Start up the Mu Editor.
4. Open up the code.py file on the Pico and modify the code you used Section XX to test the board, replacing `board.LED` with the value of the pin you connected your circuit to (see Figure XX for the pinout diagram for the Pico), keeping the `board.` prefix. For example, if you used pin GP16, you would replace `board.LED` with `board.GP16`.

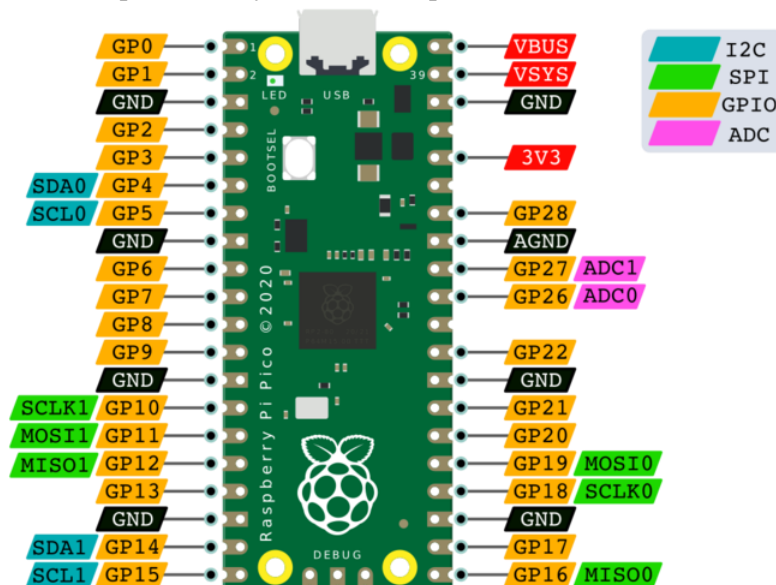


Figure 12. Raspberry Pi Pico pin designations within CircuitPython

5. Using the multimeter, measure the values and write them down next to the values you calculated. Did you notice any discrepancies between what you calculated and what you measured? If so, why do you think these arose?



If you haven't used a multimeter before or just want a refresher on how to do it, [this useful video](#) from Sparkfun contains instructions. If you don't want to watch the whole thing, the section on measuring voltage starts at 2:39, resistance starts at 5:00, and the section on measuring current begins at 6:33.

5 Implement and Test System Input

The goal of this section is to use the GPIO input to determine the state of a push button and respond by lighting up the LED.

5.1 Tools

You will need the following tools:

- Mu Editor
- Digital Multimeter

5.2 Components and Materials

You will need the following components and materials. Make sure to collect any new components and materials from your studio TAs and make sure the quantities you have collected have been recorded properly.

- a) The setup on the breadboard you created in Section 4
- b) 1x Pushbutton Switch

5.3 Reference Material

You'll need the following reference material to help you make decisions about your implementation design (links hyperlinked):

- [RP2040 Datasheet](#) [1]
- [LED Datasheet](#) [2]

5.4 Tasks

5.4.1 Review active low and pull-up resistor concepts

We've just worked with the output function of the GPIO. Now, we're going to work with the input functionality. We will be using the GPIO input essentially as a crude voltage measurement device. When the voltage measured at the pin is high (close to VCC, or 3.3V on the Pico), it registers as a "1", and when the voltage is low (close to GND, or 0V), it registers as a 0. By defining pins as inputs in CircuitPython (see the code later in this section), we enable this functionality.

An important part of using digital GPIO inputs is proper use of pull-up and pull-down resistors. Pull-up resistors are much more common, and while pull-down resistors are sometimes used, we won't cover them in this studio. Pull-up resistors are used to ensure that pins aren't left at "floating" voltages when disconnected from components, which can lead to some devices functioning improperly. [This article from SparkFun](#) provides some details on how pull-up resistors work.

Because pull-up resistors are so ubiquitous when it comes to digital microcontroller inputs, many microcontrollers (including the Raspberry Pi Pico) will actually include them built into the chip. This means that rather than having to choose resistor values and include extra external components when using inputs, the switch or other input device can be connected directly to the GPIO pin.

Now that you've read a bit about pull-up resistors, consider the circuit diagram shown in Figure 13.

1. When the button is not pressed in this circuit (open switch), what will the voltage be at the measurement point?
2. What will the voltage be when the button is pressed (closed switch)?
3. What are the digital translations (values) of these two voltages that the microcontroller will read?
4. If we removed the resistor and disconnected the measurement point from the voltage source, what would the measured value be when the button is pressed (closed)?
5. What about when it's not pushed (open)?

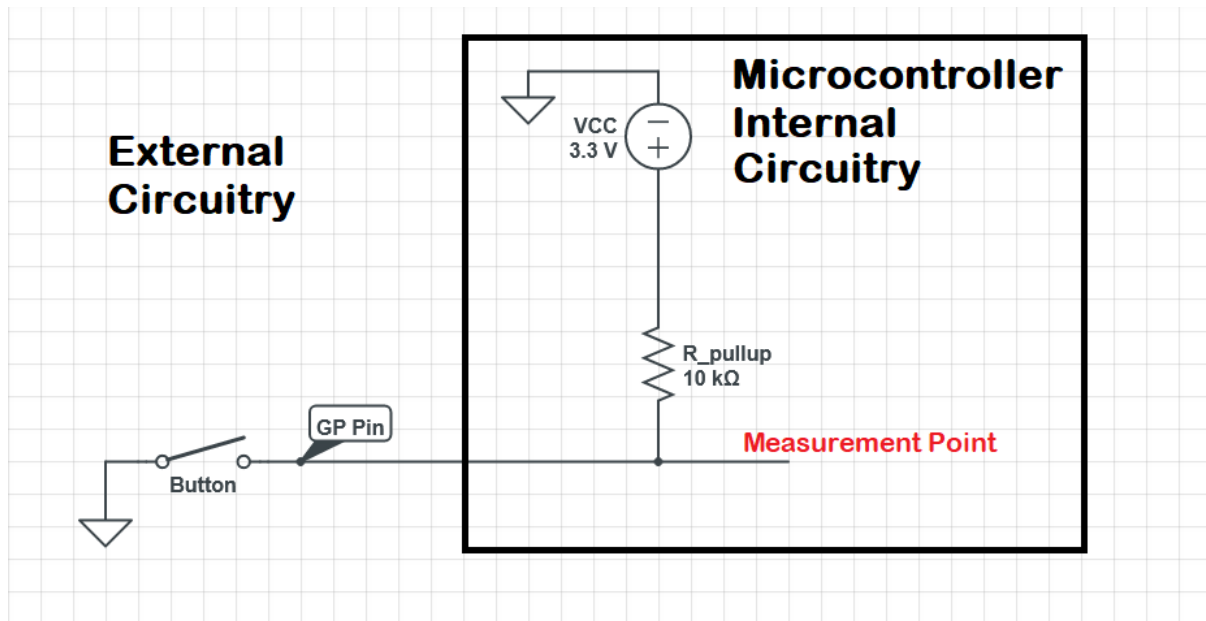


Figure 13. Internal Pull-Up Resistor circuit representation, with an external switch.

5.4.2 Build button input circuit, update code, and test

1. Power off the Raspberry Pi Pico by disconnecting the microUSB cable.
2. Leave the circuit you built in Section 4.4.3 in place.
3. Connect a switch to the breadboard as shown in Figure 14, using a different GPIO pin than the one you used for the LED.
4. Power on the Raspberry Pi Pico by connecting the microUSB cable.
5. Open the Mu Editor and replace the text in the code.py file on the Pico with the code below.
6. The LED should turn on when the button is pressed, and off when the button is released.
7. Think back to your reading from before on pull-up resistors. Why do you think there is a *not* in the last line of the code where the LED value is set?

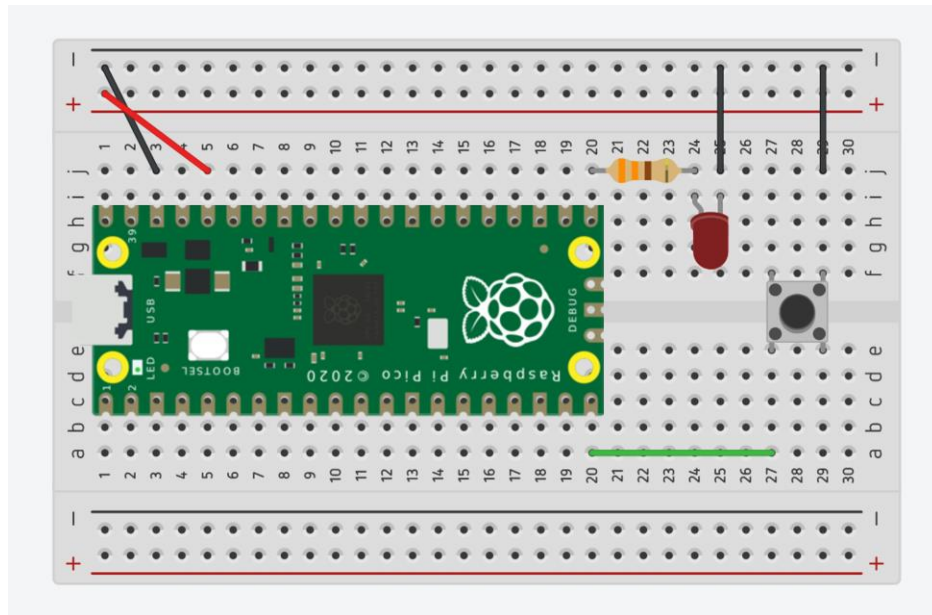


Figure 14. Input circuit breadboard picture

Full System Code

```
# Import libraries needed for blinking the LED
import board
import digitalio

# Configure the internal GPIO connected to the LED as a digital output
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

# Configure the internal GPIO connected to the button as a digital input
button = digitalio.DigitalInOut(board.GP15)
button.direction = digitalio.Direction.INPUT
button.pull = digitalio.Pull.UP # Sets the internal resistor to pull-up

# Print a message on the serial console
print('Hello there! My LED is controlled by the button.')

# Loop so the code runs continuously
while True:
    led.value = not button.value    # if the button is pressed
```

6 Challenge Scenario

The context we will use from time to time in the course to help explain concepts in context is on growing crops in colder climates with limited amount of lighting throughout the year.

Greenhouses are the technology used to control climate that crops grow in. For this studio's challenge, we will be focusing on lighting in greenhouses.

6.1 Background

For proper growth, many crops require a certain amount of light measured by the daily light integral (DLI). In northern latitudes, the DLI is below what is ideal for a growing season, so supplemental lighting is needed [3] [4]. However, according to a white paper on lighting considerations for greenhouses by Erik Runkle, professor and floriculture Extension specialist at Michigan State University, the typical light spectrum used for supplemental lighting (ratios of red and blue) produce a pink light that can be unpleasant for greenhouse workers to work in. Prof. Runkle recommends adding some white light for workers [5]. Your company designs products for the greenhouse industry and has recently taken an interest in lighting. They want to design lighting fixtures that are both crop- and worker-friendly.

6.2 Task

Your task is to design simple light fixture prototype that operates in three different modes:

1. *Off* – when the plants do not need light.
2. *Plant-only mode* – when the plants need light but the workers are not around (it would emit red and blue light)
3. *Plant-and-worker mode* – when the plants need light, but the workers are around (it would emit some white light in addition to the red and blue).

The users must be able to switch the mode of the system by pressing buttons. You have two main design options here:

1. Use a single button to switch between modes
2. Have one button dedicated to each mode.

Regardless of which approach you choose, **the system can only be in one mode at a time.**

6.2.1 Consider the abstract features of the system

6.2.1.1 *Input response behavior.*

Note that unlike the system we implemented before, the user cannot hold down the button in order to keep the LED lit, hence this system responds to *a change in value of the button*. This is illustrated in the figure below for the case of one button and one LED, where the LED turns on or off changes when the button goes from “not pressed” to “pressed”.

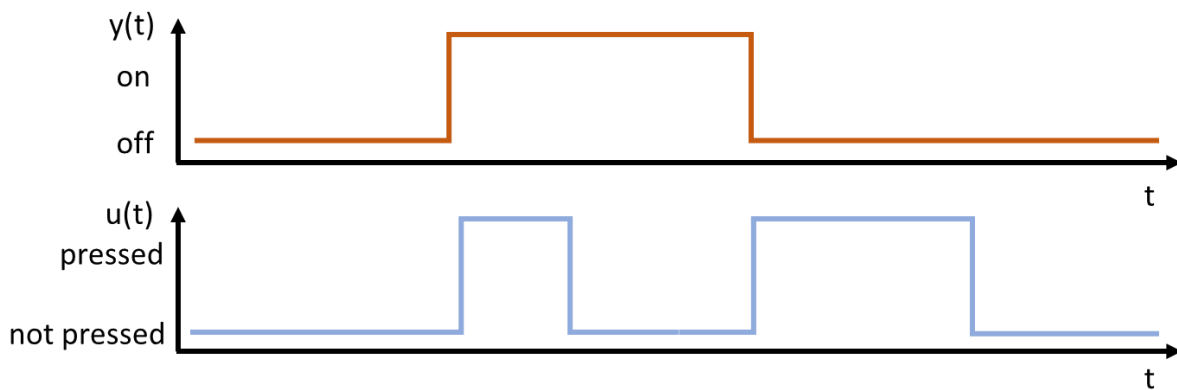


Figure 15. Timeline diagram of system input response.

Determine what the input response behavior should be for your case where you have three LEDs (blue, red, and white) and the number of buttons you decided on.

6.2.1.2 System state

Also note that unlike the system we implemented before, this system has a state. For example case of one button and one LED, there two state variables to keep track of: the state of the button, and the state of the LED. The table shows the system function $F(x(t), u(t))$. Note that the system does two things in response to the input:

1. It updates the current state to a new state (if necessary)
2. It changes the output (if necessary)

Table 2. System behavior table for $(x(t+), y(t)) = F((x(t), u(t)))$

Current State $(x(t))$		Input from button $(u(t))$	New State $(x(t+))$		Output to LED $(y(t))$
LED	Button		LED	Button	
Off	Not pressed	Not pressed	Off	Not pressed	Off
	Pressed		Off		Off
	Not Pressed	Pressed	On	Pressed	On
	Pressed		Off		Off
On	Not pressed	Not pressed	On	Not pressed	On
	Pressed		On		On
	Not Pressed	Pressed	Off	Pressed	Off
	Pressed		On		On

Using a table can get complicated quickly, especially as the number of different values the states and inputs can have increase. One representation that helps manage this complexity is the state machine representation shown below.

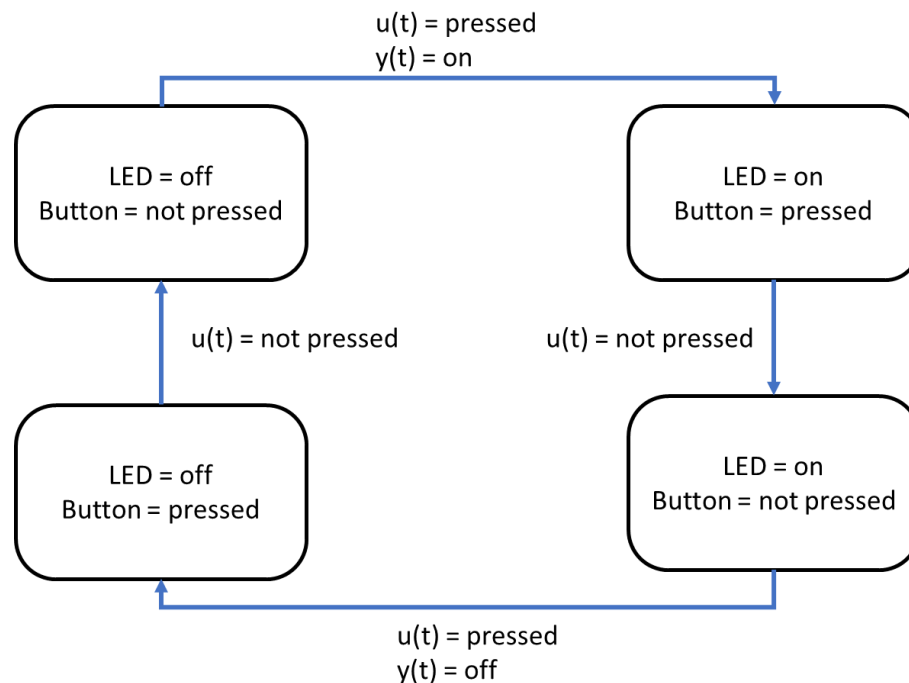


Figure 16. State machine representation of system behavior $(x(t+), y(t)) = F((x(t), u(t)))$. Boxes represent states and arrows represent conditions that cause the state to change and any change in output when state changes. Note that only changes in output are indicated.

We hope you noticed that the timeline diagram that described the input response also describes this state-like behavior.

Determine what the state response should be for your case where you have three LEDs (blue, red, and white) and the number of buttons you decided on.

Remember that you can update your design here if you find that the implementation is getting too complex.

6.2.2 Consider the System Implementation

You will need three LEDs for this system. If you do not have the exact colors, you can use others to simulate the LED. You may need more buttons, but that depends on your system design.

As in the previous sections build your circuits (don't forget about limiting the current to the LEDs)

Once you have the components, you have to modify your code from earlier for the system to behave the way you want. You'll almost certainly need to use variables, if statements, and functions, so don't forget about them (they largely work the same way as they do in regular Python).

Some of these guides from Adafruit on using CircuitPython may be useful to you.

1. [Data Structures](#)
2. [Introduction to Functions](#)
3. [State Machines](#)

6.3 Deliverables

Two items need to be uploaded to Quercus under the Assignment for this Studio:

1. A short write-up describing your design, including any diagrams or images you want to show how your design works all in one document. This should be in **PDF format**, and should be **no more than 2 pages in length**.
2. The code.py file from your Raspberry Pi Pico.

Note that you can finish this challenge outside studio and submit the deliverables later. These deliverables help us know how well you are understanding the material, and also if we need to address any gaps in understanding. We will *not* provide formal feedback on these deliverables, but we are happy to provide informal feedback. We will provide more information on how to obtain this informal feedback later.

7 References

- [1] Raspberry Pi (Trading) Ltd., "RP2040 Datasheet," 2021.
- [2] Kingbright, "Super Bright Red Solid State Lamp. Part No. WP7113SRD/D," 2007.
- [3] J. K. Craver, J. K. Boldt and R. G. Lopez, "Comparison of Supplemental Lighting Provided by High-pressure Sodium Lamps or Light-emitting Diodes for the Propagation and Finishing of Bedding Plants in a Commercial Greenhouse," *HortScience horts*, vol. 54, no. 1, pp. 52-59, 2019.
- [4] L. Pramuk and E. S. Runkle, "Photosynthetic Daily Light Integral During the Seedling Stage Influences Subsequent Growth and Flowering of Celosia, Impatiens, Salvia, Tagetes, and Viola," *HortScience*, vol. 40, no. 5, pp. 1336-1339, 2005.
- [5] E. Runkle, "Investment Considerations".