

# GETTING STARTED WITH THE RP2040



2020  
©



# Table of Contents

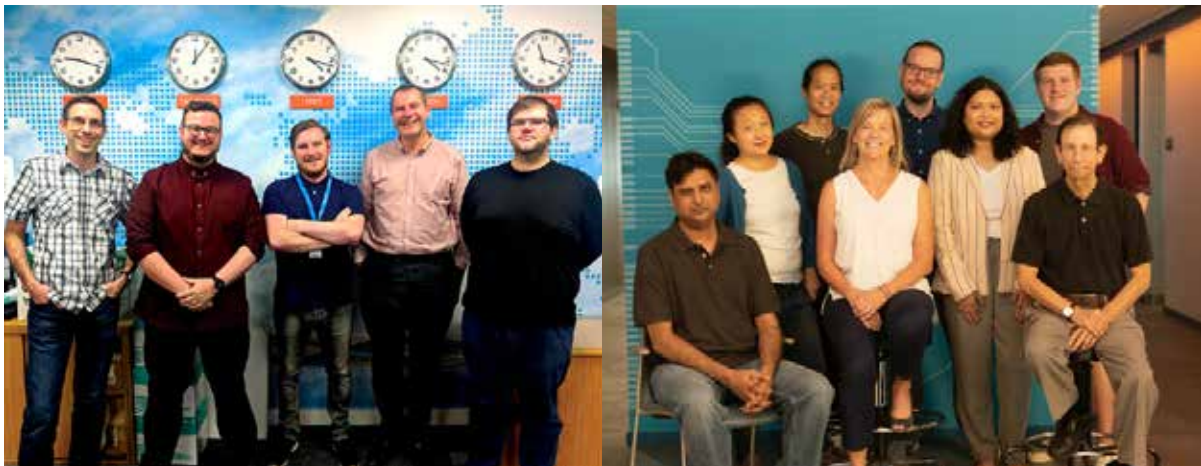
- 04**    **CHAPTER - 1**    **Introduction**
  
- 05**    **CHAPTER - 2**    **Using Thonny to Program a Device (Hello World)**
  
- 08**    **CHAPTER - 3**    **Turning an LED On/Off with a Button  
(Basic Input/Output)**
  
- 10**    **CHAPTER - 4**    **Communication with a Console  
(Basic Networking)**
  
- 13**    **CHAPTER - 5**    **Controlling a Servo (Motor Control)**
  
- 15**    **CHAPTER - 6**    **SPI Interface Example  
(Interfacing to Other Circuits)**
  
- 16**    **CHAPTER - 7**    **Related Products**

# Getting Started with the RP2040

element14 is a Community of over 800,000 makers, professional engineers, electronics enthusiasts, and everyone in between. Since our beginnings in 2009, we have provided a place to discuss electronics, get help with your designs and projects, show off your skills by building a new prototype, and much more. We also offer online learning courses such as our Essentials series, video tutorials from element14 Presents, and electronics competitions with our Design Challenges.

The RP2040 is making a big impact on the world of electronics. The RP2040 is the first microcontroller from Raspberry Pi. It's versatile, powerful, and power-efficient, making it suitable for a variety of applications, including controlling battery-powered devices. In this eBook, we'll walk you through setting up an IDE that works with the RP2040, as well as how to program a variety of simple applications on it using MicroPython.

element14 Community Team



The [RP2040](#) was first released in the beginning of 2021. It brought with it a lot of excitement, being that it was the first silicon device created by the [Raspberry Pi Foundation](#). Unlike the Raspberry Pis that are meant to run an operating system, the RP2040 is a microcontroller meant to run smaller, standalone programs. It is a device that is comparable to the Arduino and ESP microcontrollers that have also become very popular. Since the Raspberry Pi requires an operating system to run, it has many additional tasks to handle in addition to running any code to control GPIO or external devices. Microcontrollers are much lower power and less costly devices making them more suitable for interfacing with sensors and battery powered applications.

The RP2040 is being manufactured from TSMC using a 40nm process. This is an older process that also allows the device to be relatively low cost.

The RP2040 can be programmed using [MicroPython](#), C/C++, or even assembly language. Like other microcontrollers, the firmware must be written on a computer, compiled, and then loaded onto the device. Some key features of the device include the following:

- Dual ARM Cortex-M0+ that runs at 133MHz
- 264kB on-chip SRAM
- Support for up to 16MB of external flash
- DMA controller
- Interpolator and integer divider peripherals
- Two integrated PLLs for USB and core clock generation
- 30 GPIO, with 4 available ADCs
- Peripherals including:
  - 2 UARTs
  - 2 SPI controllers
  - 2 I2C controllers
  - 16 PWM channels
  - USB 1.1 controller and PHY, with host and device support
  - 8 PIO state machines

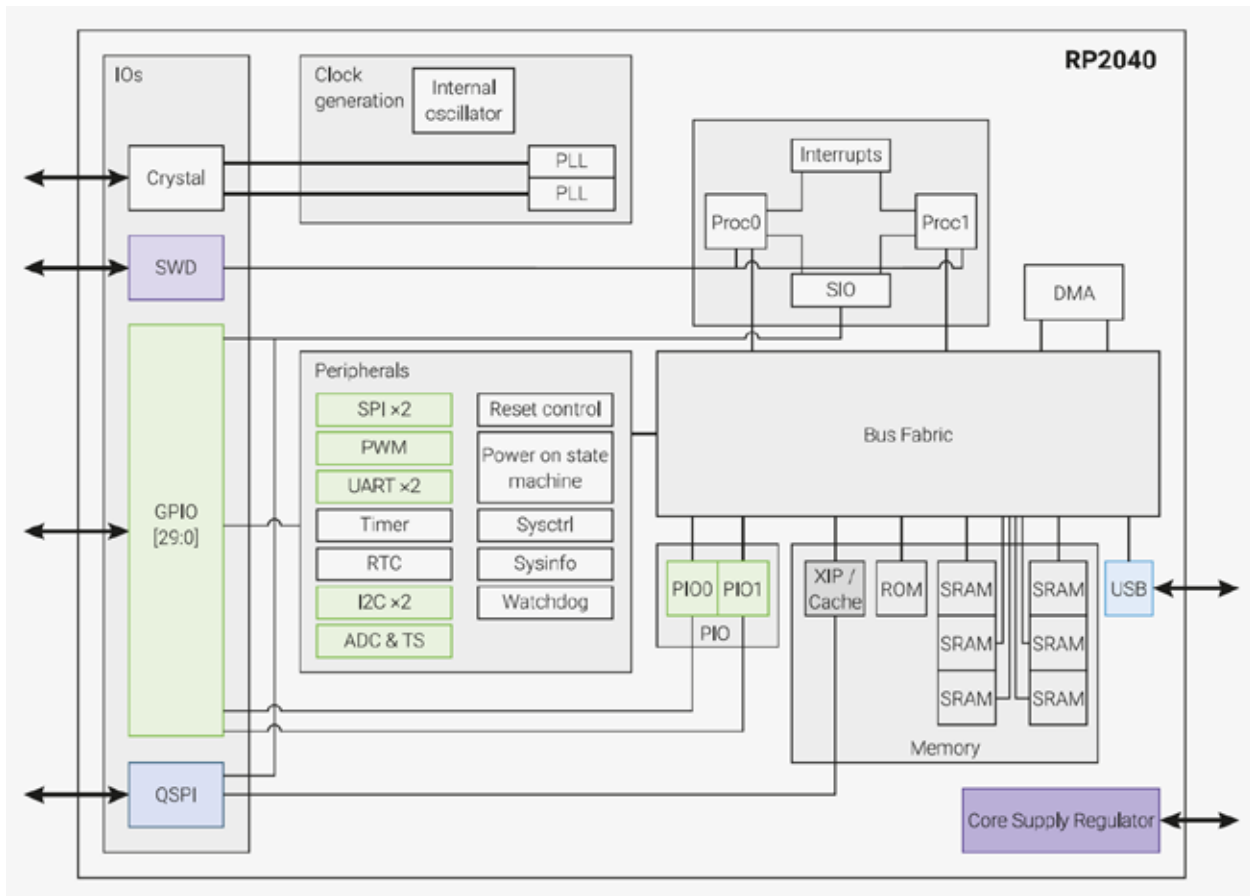


Figure 1. RP2040 block diagram. Source: <https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf>

The rest of this document will provide an overview on getting started with the RP2040. Examples will be given that demonstrate how to connect the device to peripherals and how to program the device. An RP2040 comes standard on each Raspberry Pi Pico, so we'll be using the Pico to help demonstrate the RP2040's functionality. Additionally, the Thonny IDE will be used to write firmware for the device using MicroPython. The first section will go over installing Thonny and how to program a Raspberry Pi Pico, while the rest of the sections will cover example use cases.

## CHAPTER - 2 Using Thonny to Program a Device (Hello World)

Before we can begin writing any programs for the RP2040, an IDE must be installed to write and compile code. We will use the Thonny Python IDE since it is free, simple to use, and beginner friendly. Thonny can be downloaded from <https://thonny.org>. The installation is available for Windows, Mac, and Linux machines and will take approximately five minutes to install. For this section, only a Raspberry Pi Pico board and a USB cable will be needed.



Figure 2. The Raspberry Pi Pico Board featuring the RP2040

Once the IDE is installed, we can open it and begin working towards our first “Hello World” example with the Raspberry Pi Pico. The first time we power up the Pico, we’ll need to plug it into a computer while holding down the BOOTSEL button. This puts the device in USB mass storage mode and enables us to flash the MicroPython firmware to the device. The BOOTSEL button is the only one on the Pico board and can be seen at the top left of the board in Figure 2.

After putting the device in mass storage mode, the MicroPython firmware can be installed. Clicking on the Python version in the lower right-hand corner of the Thonny IDE will bring up a selection menu. This is shown in Figure 3.

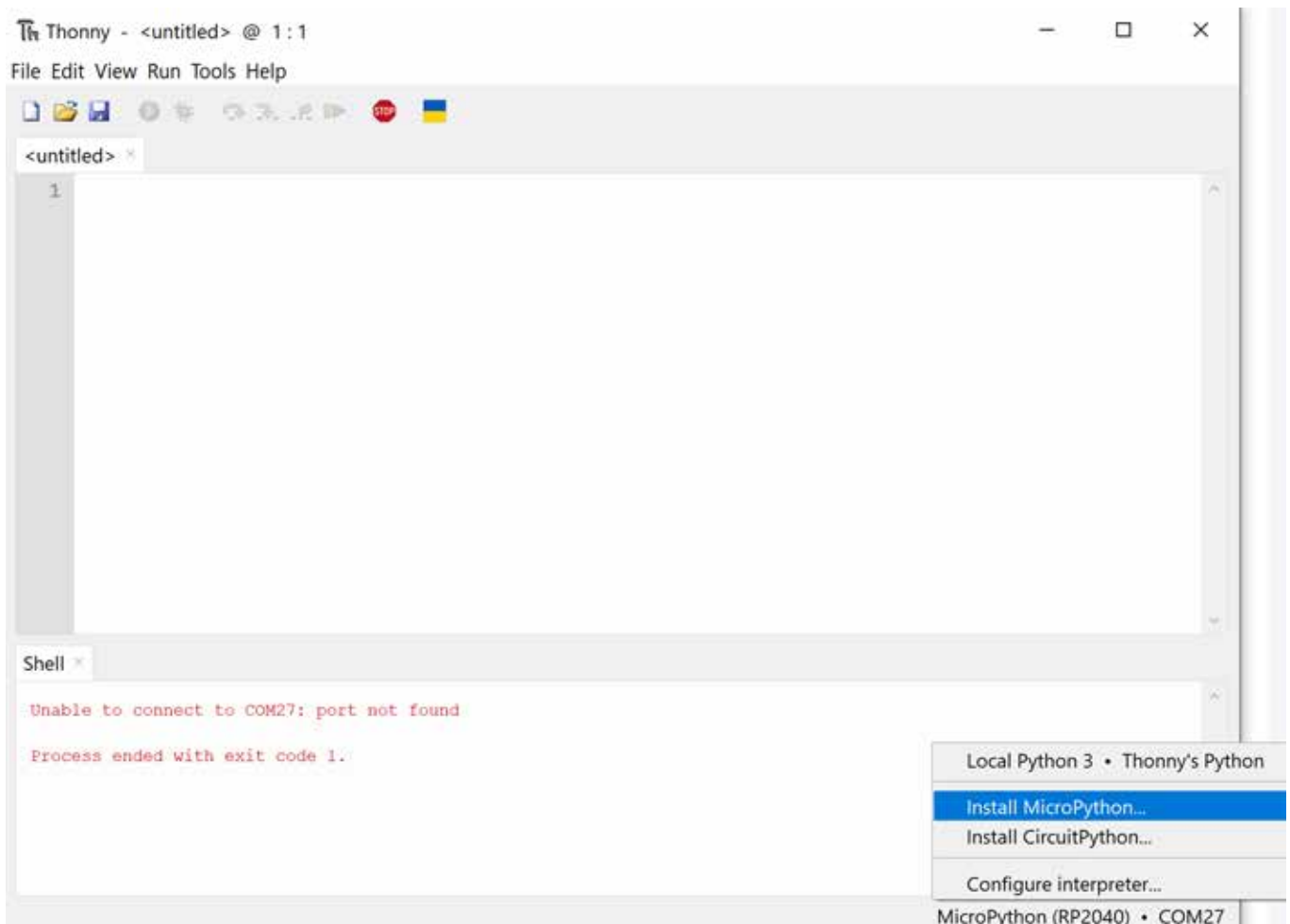


Figure 3. Installing MicroPython from Thonny using menu in lower right-hand corner

After choosing the option to install MicroPython an additional window will pop up. This window allows us to choose the variant of MicroPython to install and the version. For this example, the MicroPython variant being installed is for the Raspberry Pi Pico and upon making this selection, the version will auto-populate. We now click on install and MicroPython will be loaded onto the device within a few moments. After installation, the window can be closed. It is worth noting that we only need to install MicroPython on the device once. After this, we can simply plug the device in and begin programming it with Python.

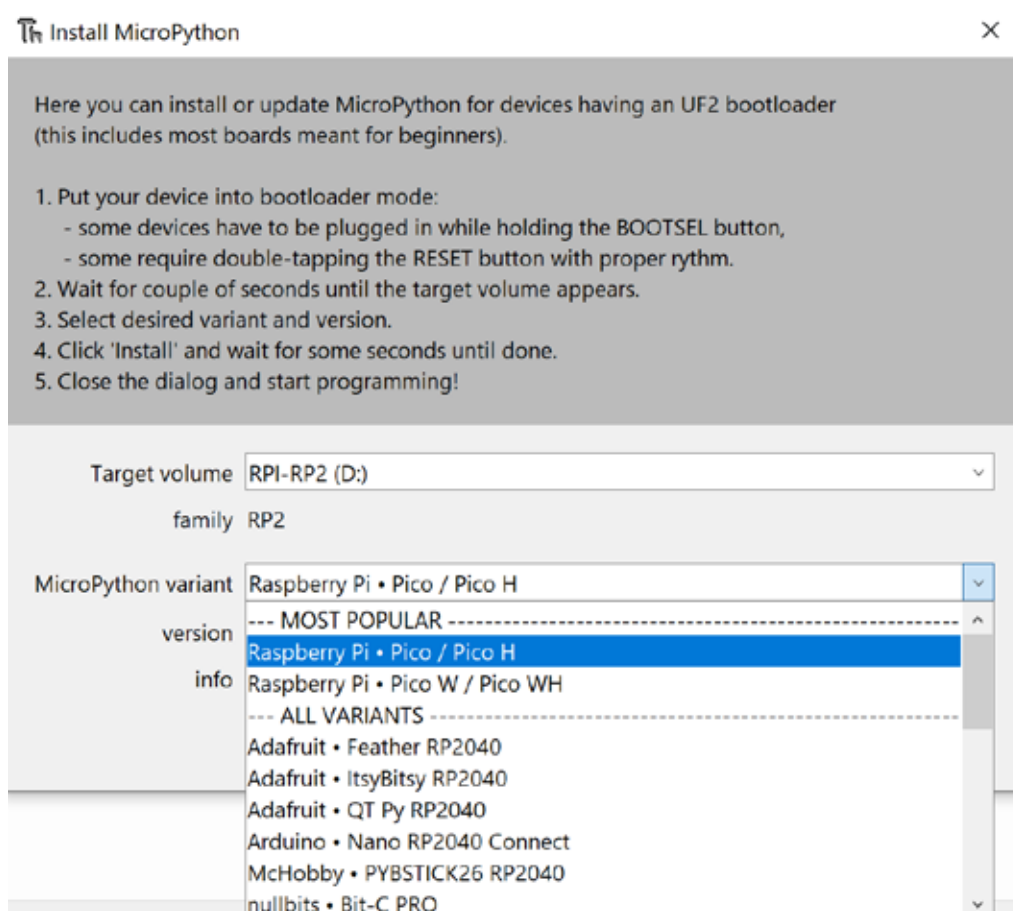


Figure 4. Additional window to choose MicroPython variant to install on RP2040

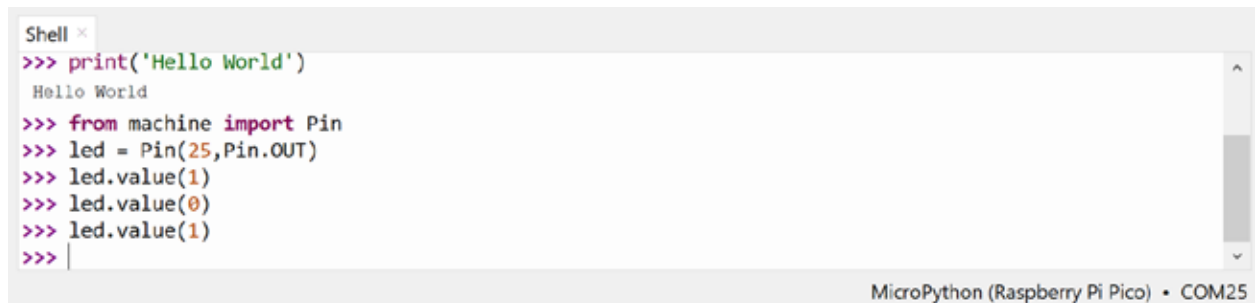
We are now able to use the shell to run Python directly on the RP2040. The shell is the small window at the bottom of the Thonny IDE where Python commands can be input and run directly. If an error message is displayed, the Raspberry Pi Pico may need to be reset by disconnecting the USB cable and reconnecting the device to the computer. In addition, if the correct COM port is not automatically updated, it may need to be selected. This is done by once again clicking in the area on the bottom of the right-hand corner of the shell where the Python version and COM port is listed.



Figure 5. Additional window to choose MicroPython variant to install on RP2040

As shown in Figure 5, we are able to test the functionality of the firmware and connectivity of the device by simply running a `print("Hello World")` into the shell. This code will run directly on the RP2040. Furthermore, we can now test out the “Hello World” of microcontrollers by blinking the Pico’s on-board LED.

Using MicroPython, import a module named `machine`. This module contains specific functions for the RP2040 that allow us to easily control the device. Using the `machine` module, we will blink the on-board LED, which is connected to GPIO 25. Entering the code as shown in Figure 6 directly into the shell will enable us to blink the on-board LED.

A screenshot of a MicroPython shell window titled "Shell x". The window contains the following code and output:

```
>>> print('Hello World')
Hello World
>>> from machine import Pin
>>> led = Pin(25,Pin.OUT)
>>> led.value(1)
>>> led.value(0)
>>> led.value(1)
>>> |
```

The status bar at the bottom right of the window reads "MicroPython (Raspberry Pi Pico) • COM25".

Figure 6. Code showing how to blink the on-board LED through the shell

Turning the LED on is as simple as writing three lines of code to the device. After initializing the correct pin as an output, we simply type a “1” to turn it on, or a “0” to turn it off.

Sending the RP2040 commands is quick and simple using the Thonny IDE and Pico. In the following sections, we will write programs using the text editor in Thonny and load them onto the Pico device. When developing code for an application or project this will be the general method used to program the device.

## CHAPTER - 3

## Turning an LED On/Off with a Button (Basic Input/Output)

In this section, we will demonstrate how to use GPIO for simple input and output. This is some of the most basic functionality that can be done on a microcontroller. For this specific example, we will read the input to a pin that is connected to a switch. If the button is pressed, we will turn on an LED, otherwise, the LED will remain off. To build and test this circuit, see the required components list. It also helps to solder headers onto the Pico so that it can be inserted into a breadboard for prototyping and testing. Alternatively, a Pico can be purchased with pre-installed headers.

### Required Components List:

- Raspberry Pi Pico
- Breadboard
- Jumper wires
- USB Micro Cable
- LED and current limiting resistor (~470  $\Omega$ )
- Tactile breadboard button

[See Related Products section](#)



It also helps to know the pinout on the Pico, so that we know where the GPIO pins are located. The pinout for the Pico is shown in Figure 7.

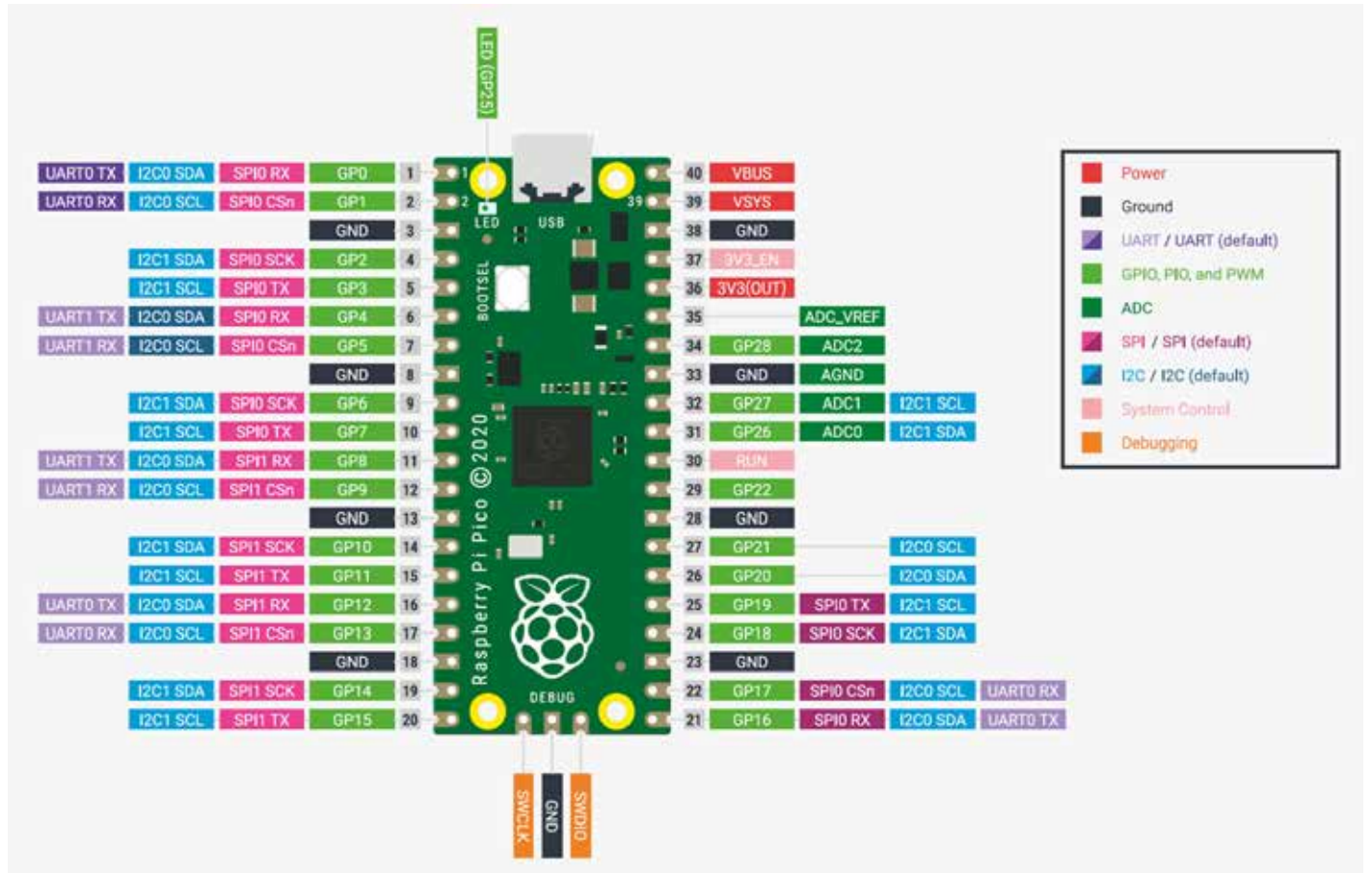


Figure 7. Pinout diagram for the Raspberry Pi Pico board

For this example, we will connect the LED connected from GP21 to Ground through a current limiting resistor (~470  $\Omega$ ), so that when we set GP21 high, the LED will turn on. In addition, we will connect the tactile button to GP14. We add a pull up resistor (~4.99k) at the node connected to GP14, so that it will appear as a high voltage when the button is not pressed. The other side of the button will connect directly to ground. As a result, when the button is pressed, the voltage seen at the input of GP14 will go from high to low. The connections for this can be seen in the diagram below.

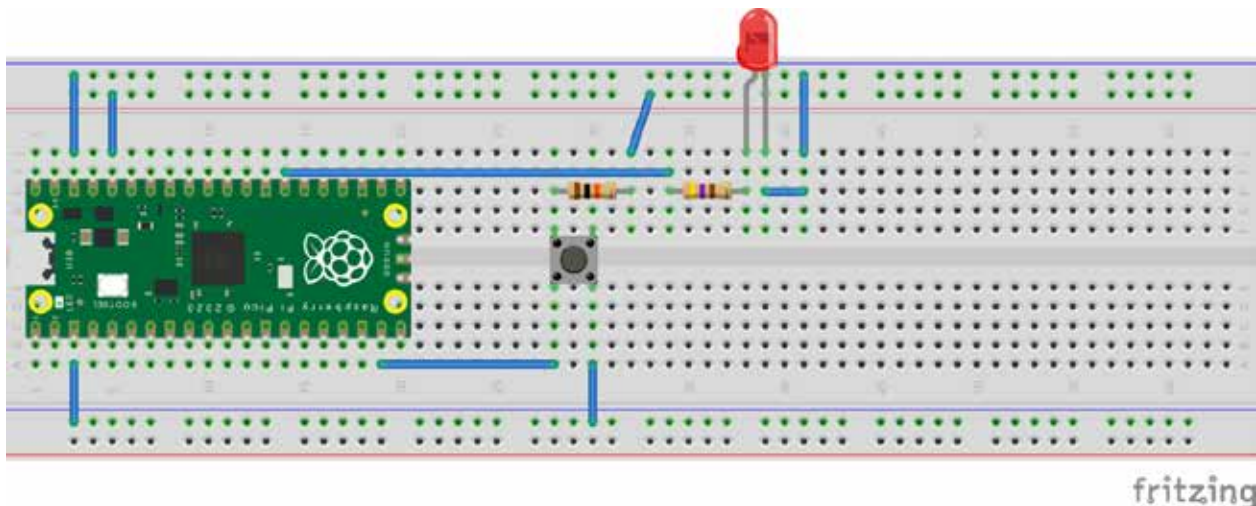


Figure 8. Connections needed for basic input and output example

With the hardware now connected, we can begin writing the code for the device. The code for this example is shown in Figure 9. Two modules are required. First is the *machine* module, which allows us to define pins as inputs or outputs and read or write to them. The second module we will import is called *time*; this is used to add delays into the program.

After the modules are imported, the device should be initialized. GP21 is set as an output and GP14 as an input. This is done in lines 4 and 5. Note that when defining a pin as an input there is an optional argument that can be added to add an internal pull up or pull down on the pin. However, since we have the pull up added externally, we can omit this argument. For the final part, an infinite loop is created that will continuously read the state of the voltage on GP14 (line 7). If the button is pressed, the voltage will go low (0V). When this happens, we want to set the *led.value* to 1 (or high). This is shown in line 9. In all other circumstances, the output will be low and the LED will be turned off. Note that a one second delay is added into the code after setting the LED high. This keeps the LED from blinking sporadically and allows us to easily observe the LED turning on.



```
[main.py] x
1 from machine import Pin
2 import time
3
4 led = Pin(21,Pin.OUT)
5 button = Pin(14,Pin.IN)
6
7 while True:
8     if button.value() == 0:
9         led.value(1)
10        time.sleep(1)
11    else:
12        led.value(0)
13
14
```

Figure 9. Code for basic input and output example

There are a couple of different ways to run the code on the Pico device. We can press the green button in the top task bar (green circle with white arrow), or we can save the code directly onto the Pico by going to **File** at the top left, then choosing **Save as**, and finally choosing the Raspberry Pi Pico as the target device. If the program is run using the green button in the task bar, it will have to be explicitly stopped using the red stop button in the same task bar before additional code can be loaded or the shell used. In addition, code run using the green run button is not saved onto the Pico. To get the code to run after a power cycle the program must be saved onto the Pico as *“main.py.”*

## CHAPTER - 4

## Communication with a Console (Basic Networking)

The next example will cover how to work with a serial console over the USB connection on the Pico. A serial console can be useful for debugging or controlling the Pico through your computer. For instance, if you want to use your computer as the main method of communicating with the device, a serial console is an ideal

choice. If you are developing a project and want to print output to a screen for debugging, the serial console is again an easy and simple way of doing so. There are many free options available, including [PuTTY](#), [Tera Term](#), and [Minicom](#).

For this example, we will use a serial console (Tera Term) to read in user input and turn on or off an LED depending upon the user input. The Pico board alone can be used for this example since it has a single LED on board that can be controlled. We can also use the same circuit as in the previous example, but without the button. The hardware for this includes ([See Related Products section](#)):

- Breadboard
- Raspberry Pi Pico
- Jumper wires
- USB Micro Cable
- LED and current limiting resistor (~470  $\Omega$ )

For this example, we will use the same circuit as in the prior example without the button. This set is depicted in Figure 10.

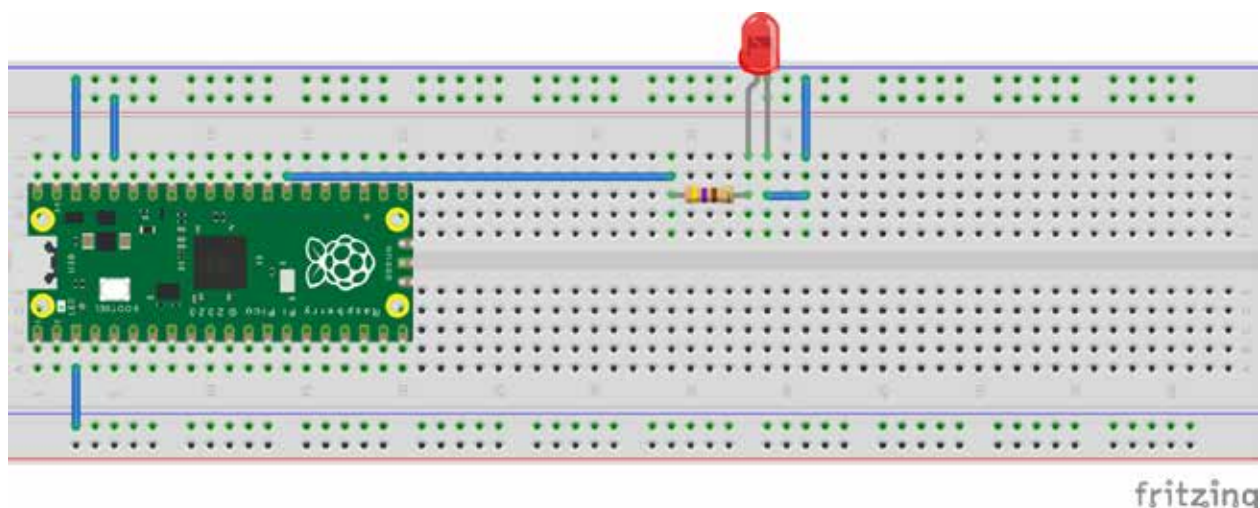


Figure 10. Circuit used for serial console communication example

Two additional modules will be used (*machine* and *time* are still required): the *sys* module and the *select* module. Both modules are subsets of the CPython module and make working with the serial console simple. The code written for this example is shown in Figure 11.

We start by importing the required modules. Next, we define the LED pin and set it as an output. The infinite loop comes next; this ensures that the program will continue to run for as long as the Pico is powered on. Within the infinite loop is an additional *while* loop that waits for user input from the serial console using the *sys* and *select* libraries. Note that in this specific example, we are only reading in one character. There are additional ways to read in all characters; however, that is beyond the scope of this example.

Any input read in from the serial console is represented by a hexadecimal number. This hexadecimal number represents an ASCII value. To correctly decode it, we first need to encode it using UTF-8 and then decode it as an ASCII character. This will save a string into a variable (line 15) that we can use as needed.

```
File Edit View Run Tools Help
rp2040_code_ex_02.py
1 from machine import Pin
2 import machine
3 import time
4 import sys
5 import select
6
7 led = Pin(21,Pin.OUT)
8
9
10 while True:
11
12     while sys.stdin in select.select([sys.stdin], [], [], 0)[0]:
13         ch = sys.stdin.read(1)
14         print("User input is: ",str(ch.encode('utf-8').decode('ascii')))
15         data = ch.encode('utf-8').decode('ascii')
16         if data == '1':
17             led.value(1)
18             print("Turning LED ON")
19         elif data == '0':
20             print("Turning LED OFF")
21             led.value(0)
22         else:
23             print("invalid input to control LED")
24
```

Figure 11. Code written to communicate with serial console and turn on or off an LED

The rest of the code consists of simple *if* and *else* statements that compare the user input value to a “1” and a “0”, turning the LED on or off correspondingly. After saving this code to a Pico device, we can then open a terminal in order to control the LED. An example of the control using Tera Term is illustrated in Figure 12.

As mentioned before, the serial terminal can help when working through a design or with debugging. For example, when working with a temperature sensor, we can print the calculated temperature to the screen to determine if the equations are accurate before integrating it into a larger system. Furthermore, when designing a circuit that incorporates an analog-digital converter (ADC) or digital-analog converter (DAC), having real time feedback and control through a computer can have major benefits.

```
COM14 - Tera Term VT
File Edit Setup Control Window Help
User input is: 1
Turning LED ON
User input is: 0
Turning LED OFF
User input is: 2
invalid input to control LED
```

Figure 12. Serial terminal used to communicate and control an LED through the Pico

This example will cover how to control a servo using a pulse width modulated (PWM) output from the Pico device. Servos are used in many applications, especially robotics. They are made up of three core key pieces that include a DC motor, a controller circuit, and a potentiometer or some other sort of device for feedback. The DC motor drives gears that determine the speed, torque, and position of the output shaft. Almost all hobbyist servo motors have a 3-pin header for power and control. Generally, there are a black and red wire for ground and DC power, respectively, and an additional wire that may be

yellow or white for controlling the position. Servos come in many shapes, sizes, and power handling capabilities; thus, it is important to make sure the device is compatible with the Pico before using. The hardware needed for this example is as follows:

- Breadboard
- Raspberry Pi Pico
- Jumper wires
- USB Micro Cable
- Hobby Servo

[See Related Products section](#)

To begin, we connect the servo as shown in Figure 13. We will once again use GP21 for our control pin, although any of the pins can support a PWM output. A PWM signal is a digital signal that changes states from high to low. The percentage of time the signal is high over its period is known as the duty cycle. Thus, a 50% duty cycle signal will spend half of the time high and half of the time low. As a result, the average DC power output from the pin is half the rail voltage. Taking advantage of this technique, we can use a PWM signal to produce a variable voltage output.

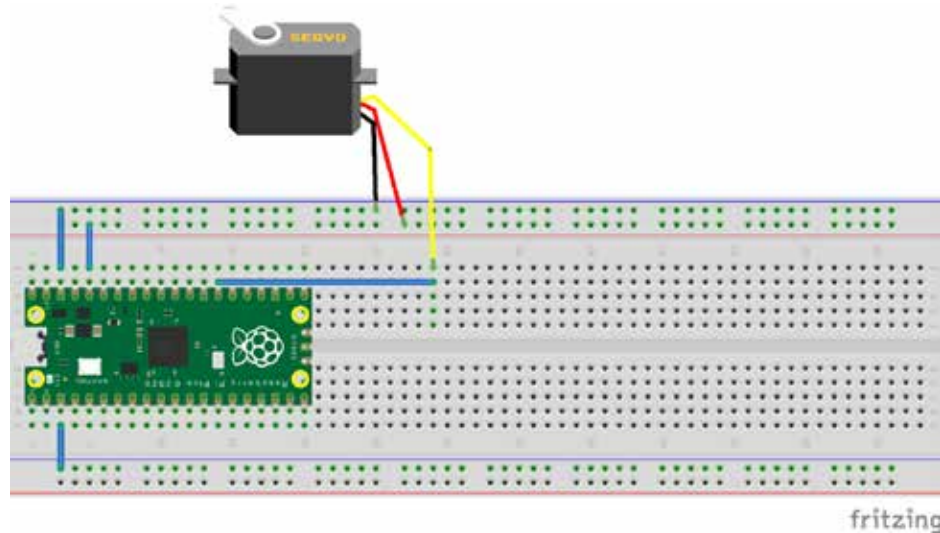


Figure 13. Circuit connections for servo control example

It is important to understand how a servo is controlled before jumping in. All servos will have a minimum and maximum pulse width which they will respond to. Outside of this range, they will not respond and may stop holding their position. Different servos will have different specifications, but for this example, let's assume this range: a minimum pulse width is approximately 620  $\mu\text{s}$ , which will set it at a 0° angle, a maximum pulse width is 2420  $\mu\text{s}$ , which will set it at a 180° angle, and to achieve a 90° angle a pulse width of 1520  $\mu\text{s}$  is needed. Figure 14 helps visualize the PWM signal and pulse width. Note that 20ms is the period of a 50Hz frequency which is what we will set the PWM frequency to.

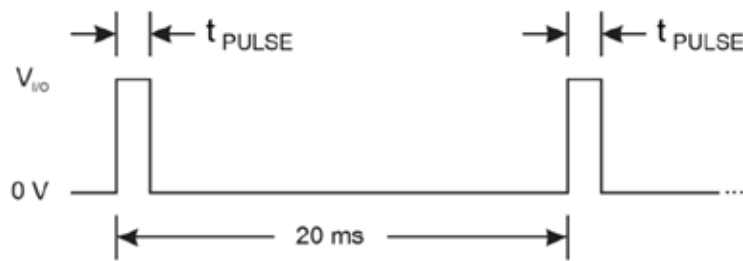


Figure 14. PWM waveform

Now that we understand what a PWM signal is and how we can use it to control a servo, we can begin writing the code. The code for this example is shown in Figure 15.

```

File Edit View Run Tools Help
rp2040_code_ex_03.py * x
1 import machine
2 import time
3
4 servoPin = machine.Pin(21)
5 servo = machine.PWM(servoPin)
6 servo.freq(50)
7
8 while True:
9     servo.duty_ns(620000) # 0 degrees
10    time.sleep(2)
11    servo.duty_ns(1520000) # 90 degrees
12    time.sleep(2)
13    servo.duty_ns(2420000) # 180 degrees
14    time.sleep(2)
15
16

```

Figure 15. Code for controlling a servo example

In the code for this example, we simply initialize the Pico and set up an infinite loop that will change the position of the servo every two seconds. To initialize the device, we first need to import the *machine* and *time* libraries. We then define the pin that we will use to control the servo (GP21). After that, we create a *servo* object that will use a PWM output and define the PWM frequency as 50Hz (lines 5 and 6). The next steps consist of creating the infinite *while* loop, setting the servo positions, and adding a delay. To change the duty cycle or pulse width the *duty\_ns* function is used. This allows us to change

the duty cycle by specifying a desired pulse width in nanoseconds. The pulse width for specific settings of the servo can be found in the servo's datasheet.

After loading the code onto the Pico device, the servo will change locations every two seconds while the device is powered on. Generally, some sort of input will be read to control the servo position. This can be a controller, potentiometer, or depending upon the end application, a sensor.

When working with more complex integrated circuits, usually some standard communication interface is utilized. For instance, when controlling an LED or switch, only a GPIO may be required; however, some circuits can output data or may have various settings that need to be configured for expected operation. In these cases, an SPI, I2C, or even a UART interface may be used. Learning how to work with these various interfaces can greatly increase the types of circuits you can utilize for your designs.

The RP2040 has built-in communication interfaces that make communication through these interfaces

simple. For this example, we will demonstrate generic code to create a simple SPI interface in the RP2040 using a Raspberry Pi Pico. There will be no hardware required for this example.

The only module required to import for a SPI interface is the *machine* module. Using the *machine* module, we can set up the clock, transmit, receive, and chip select pins needed for a SPI interface. Next, we need to initialize the SPI interface and write a function to handle the data transfer. The code for this can be seen in Figure 16.



```
File Edit View Run Tools Help
rp2040_code_ex_04.py x
1 import machine
2
3 clk = machine.Pin(2)
4 tx = machine.Pin(3)
5 rx = machine.Pin(4)
6 cs = machine.Pin(5)
7
8 spi = machine.SPI(0, baudrate=1000000, polarity=0, phase=0, sck=clk, mosi=tx, miso=rx)
9 cs.init(machine.Pin.OUT, value = 1)
10
11 def spiTransaction(data):
12     cs.value(0)
13     output = spi.write_readinto(data, None)
14     cs.value(1)
15     return result
16
```

Figure 16. Code for generating a SPI interface

First, we import the *machine* module and then begin initializing the SPI interface by creating variables for the pins to be used. Note that for this simple SPI initialization, the proper pins must be selected. The pinout diagrams for the Pico as well as the RP2040 has them identified and they can also be found in Figure 7 of this eBook. Lines 8 and 9 are the final lines needed for initializing the SPI. In line 8 the baud rate (or clock rate) of the SPI interface is selected, in addition to the polarity and phase. These will

vary depending on the device being used, but the polarity describes the idle state of the clock when no transaction is occurring, and phase tells us whether data begins to be captured on the rising or falling edge of the clock.

The last part of the code (lines 11 – 15) creates a function that can then be called to read or write to the device. In the function, *cs* (chip select) is pulled low, and the *write* transaction occurs, after which

**cs** is pulled back high to complete the SPI transaction. If you are trying to read from the device, this function will still work. The `spi.write_readinto` command will return values for a read transaction. Thus, to communicate with a device using the example code given, we would only need to call the function defined, passing the needed data. The data format used by SPI interfaces usually includes a read/write bit, address, and data (if writing to the device). For reading from a device, the data will only consist of a read/write bit and address to be read.


## CHAPTER - 7

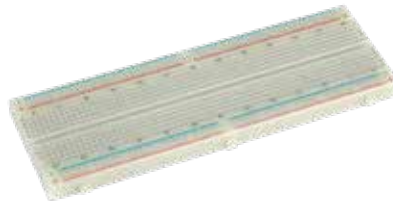
## Related Products

Overall, this eBook has introduced the RP2040 microcontroller. There are many additional areas to explore with the RP2040; these simple examples are just the beginning of what is possible.

 Raspberry Pi Pico



 Breadboard



 Jumper Wires




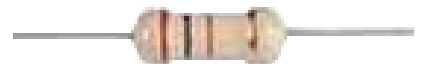
 USB Micro Cable



 LED Bulb



 Current Limiting  
470  $\Omega$  Resistor



 Tactile Breadboard  
Button



 Pull up 4.99 k $\Omega$   
Resistor



 Hobby Servo





See more about [Raspberry Pi](#) on the element14 Community.



element14  
/ AN AVNET COMMUNITY

300 S. Riverside Plaza, Suite 2200  
Chicago, IL 60606

<https://community.element14.com/>



[Facebook.com/e14Community](https://www.facebook.com/e14Community)  
[Twitter.com/e14Community](https://twitter.com/e14Community)