Lift-Off with CodeX

Learn Python fundamentals through fun projects with the CodeX.

Mission 1 - Welcome to Lift-Off Peripherals!

Welcome Back!

Are you ready to expand your coding knowledge? I'll be your guide as **YOU** learn to interact with external components.

You will complete some *Outer Space*-themed projects in your mission to *Lift-Off* with **code**.

Why add peripherals?

- A CodeX is great, but sometimes you just need *more!*
- Sometimes you need some hardware peripherals.
- Like an **LED**, a **sensor**, a **pump**, or a **servo**...
- Peripherals allow you to interact with the world in **new** and **exciting** ways.

And now... It is time for YOU to hook up your favorite peripherals and get coding.

As you complete this *project-based* course, you'll be learning skills that will allow you to imagine and create *ANYTHING*!

Objective 1 - Mission "Peripherals"

Mission Briefing:

A crew of astronauts is preparing to embark on a *long* and *arduous* journey into the vast reaches of **outer space**.

The crew must launch from *Earth*, survive a months-long journey through space, and then establish a colony on *Mars*.

They will need many electronic applications to aid them on their journey, and they are looking for *your* help to build them!

Each project in the Lift-Off Peripherals course will *simulate* real-world electronic applications that could be used:

- On-board a space shuttle
- As part of mission control
- On the *Mars* colony

Your **mission** is to utilize the **CodeX** with different peripherals to build applications needed by the crew.

The CodeX has four expansion ports for connecting peripherals.

- Each PORT is located just above a NeoPixel.
- Notice that the number for each pixel also points to the PORT.
- **PORT0** is just above **pixel 0** and is to the left of the CodeX.
- **PORT3** is just above **pixel 3** and is to the right of the CodeX.

Concept: *Python with CodeX Extension*

The **Peripherals** course is intended to be a *follow-on* to the **Python with CodeX** curriculum.

- You will need to **recall** and **apply** some basic *Python* knowledge learned in that course.
- Don't worry if you are a little rusty. You will get plenty of refresher along the way!
- Check the Hints for the recommended *Python* knowledge base.
- If you already have those *Python* fundamentals then this course will be a **snap**!

Hint:

- ## Recommended *Python* knowledge base:
  - 🔧Variables
  - 🔧Functions and 🔧parameters
  - 🔧Branching with `if`, `elif` and `else`
  - `while` 🔧loops
  - 🔧Timing using the `sleep` function
  - Using 🔧comments
  - 🔧Comparison operators `==`, `!=`, `<`, and `>`
  - 🔧Boolean values `True` and `False`

Goal:

- Click on **PORT0** on the CodeX in the 3D scene!
  - *Hint:* Rotate the CodeX to the **side** view.

Tools Found: CPU and Peripherals

Solution:

N/A

Objective 2 **- Explore Your Kit!**

Explore your Kit!

The LiftOff Peripherals Kit contains many different 🔧peripherals. There are inputs and outputs, buttons, sensors, servos, and even a pump!

Check the 💡 Hints for details about each peripheral.

Physical Interaction: *How do I find this information again?*

Click this link → 🔧LiftOff Kit

This will add a topic to the **Tool Box** in the *lower right corner* of your screen. The tool box allows you to return to **important** topics. You can access general information on the peripheral kit and learn even **more** about each of the components in your kit.

Hints:

- ## Button

  A standard momentary push button. Buttons are used in a wide variety of applications to take user input.

- **Switch**

  This switch locks into place when pressed. Switches are generally used to apply power or change a setting.

- **Potentiometer**

  Often referred to as a knob. The potentiometer can be physically turned. It returns a value based on the position it is set to.

- **Microswitch**

The microswitch is used in a variety of applications. Operationally, it is nothing more than a button. It can be used as a crash sensor to detect robot touch.

- ## Motion Sensor

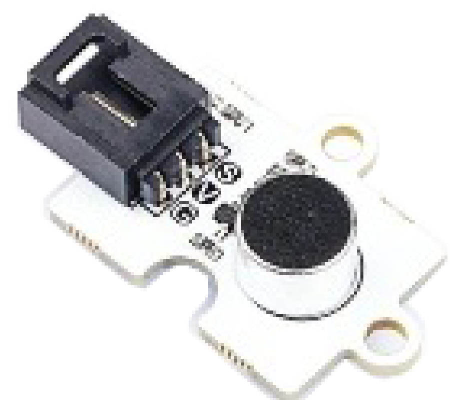
A PIR (Passive Infrared) Sensor measures IR light radiating from objects. Typically used in applications such as motion lights and alarms.

- ## Temperature Sensor

Can read a raw value of the temperature. Requires some calibration to provide a specific temperature value in Celsius or Farenheit.

- ## Sound Sensor

  This sensor is sensitive to sound intensity. It can be used to detect noises.



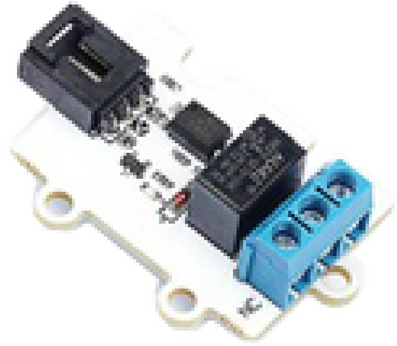
- ## Light Sensor

  The photocell is used to measure the amount of ambient light. It has many uses such as smoke detection, solar monitoring, and light dimming.

- ## 3V Relay

  Can be used to switch larger power and voltages to devices. Could be used for a DC Motor or an AC Lamp.

  Contact Ratings: 2A 120VAC / 2A 24VDC



- ## Red LED

A red LED (Light Emitting Diode). Outputs a red light.

- A **white** LED is also included in the kit.

- ## 8 RGB LED Ring

8 RGB (Red Green Blue) LED Ring. Also called a NeoPixel ring. This device has 8 individual RGB LED pixels. Pixels can be illuminated any color and assigned individually.

- ## Object Sensor

Sometimes referred to as a Hunt Sensor. It contains an LED that emits Infrared (IR) light, and a Phototransistor that detects IR light reflected from nearby objects. It hs many uses such as object detection and line following.

- ## 180 Servo

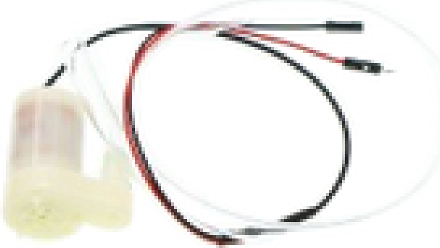
More commonly called a positional servo. This servo can rotate clockwise or counterclockwise 180 degrees. It can be used to move project components to a specific location.

- ## 360 Servo

  More commonly called a continuous rotational servo. This servo can rotate 360 degrees. It operates continuously in the clockwise (forward) or counterclockwise (backward) direction.

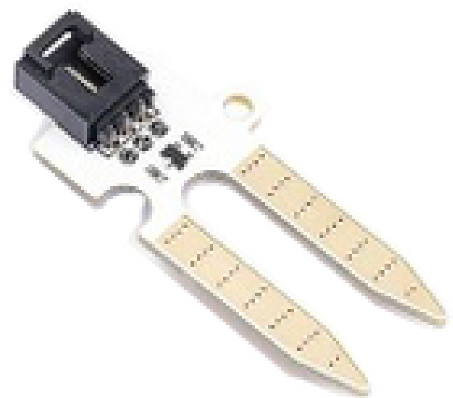
- ## Water Pump

  This is a small water pump. It is designed for vertical, submersible operations and can be used to transfer water from one location to another.



- ## Soil Moisture Sensor

  Detects the amount of moisture present in the soil surrounding it. Most often used to determine when a plant needs water.



Goal:

- Click on LiftOff Kit to add the topic to your toolbox.

Tools Found: CPU and Peripherals, LiftOff Kit

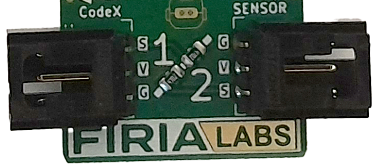
Solution:

N/A

Objective 3 - More Peripherals

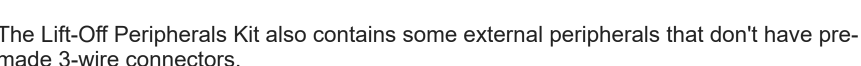
But wait ... there's more!

The Lift-Off Peripherals Kit contains a few more peripherals.



One of them is the **divider**.

- The divider is a circuit that cuts in half the voltage from analog sensors.
- It enables the analog sensors to get their full range.
- The **divider board** is first connected to the CodeX, and the **analog sensor** connects to the **divider board**.



The Lift-Off Peripherals Kit also contains some external peripherals that don't have pre-made 3-wire connectors.

- The CodeX has a pair of expansion ports, located on the front near the bottom of the board.
- The external peripherals are connected on the breadboard, which is mounted on the CodeX using the expansion ports.
- Use jumper wires to connect the periperals with connection points.

100-Ohm Resistors

Red and Amber LEDs

Mini Breadboard

HC-SR04 Ultrasonic Sensor

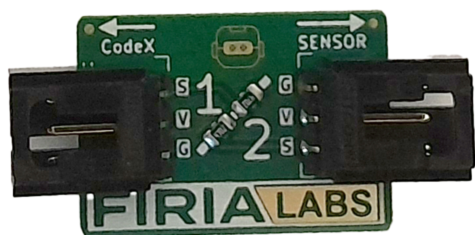
Jumper Wires

Check the 💡 Hints for details about the additional peripherals.

Hints:

- ## Divider Board

  The analog sensors can put out 5V, but the CodeX ADC can only show around 2.8V. The divider board cuts the voltage in half so the sensors can read their full range.



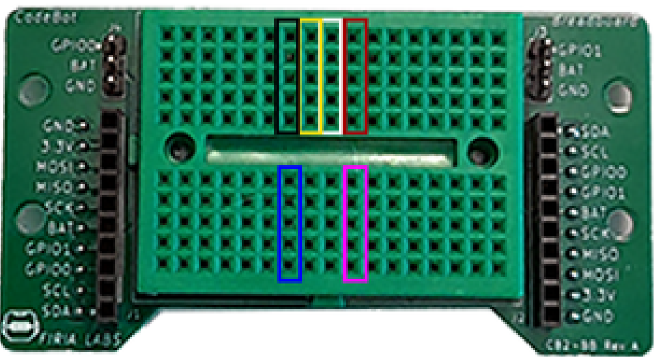
Analog sensors in the Kit include:

- Potentiometer
- Temperature Sensor
- Light Sensor
- Sound Sensor
- Soil Moisture Sensor

- ## Mini Breadboard

  The mini breadboard is one of the smallest breadboards available. The tiny holes on the breadboard let you easily insert electronic components to build a circuit.

  - The components can be external peripherals like LEDs, sensors, buttons, and speakers.
  - The connections are not permanent, so it is easy to remove components and reuse the breadboard for a different project.
  - The CodeX will supply power to the mounted breadboard, which can then power the added components.

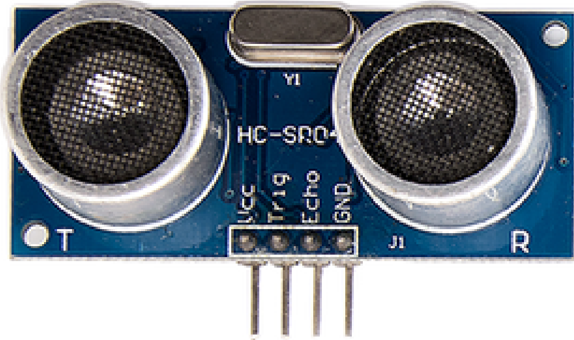


- ## Jumper Wires

  Jumper wires are used to make connections on a breadboard. Their ends are easy to push into the breadboard holes. The wires come in varying colors. It doesn't matter which colors you use, but color coding the project helps you stay organized and keep track of what you are connecting.

- ## HC-SR04 Ultrasonic Sensor

  The ultrasonic sensor uses sonar to determine the distance to an object. The sensor returns the time it took to give and receive the signal.



- ## LEDs and Resistors

  The LEDs that will connect to the breadboard will need resistors when they are built into a circuit. LEDs don't respond well when voltage fluctuates, and this can damage the LED.

  - The **positive** lead on the LED is the long side.
  - The **negative** lead on the LED is the short side.

- The resistor doesn't have a designated **positive** or **negative** lead and can be connected in any order.

Goal:

- Rotate the CodeX in the 3D scene.
  - Click on one of the **Expansion Ports** on the CodeX.

Tools Found: CPU and Peripherals

Solution:

N/A

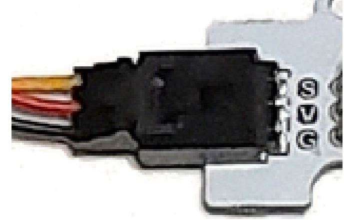
Objective 4 **- Connecting a Peripheral**

Make some connections!

The **CodeX** makes connecting peripherals easy. Here is what you need to know:

The peripheral connectors are labeled with **S**, **V** and **G**. Connect a cable to the peripheral this way:

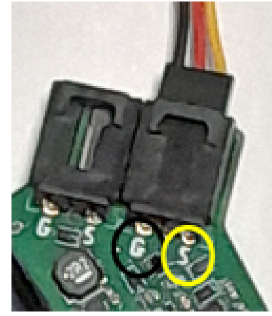
- Yellow is **S (Signal)** and is connected to the **S**
- Red is **VCC (Power)** and is connected to the **V**
- Black is **G (Ground)** and is connected to the **G**



The two ends of the cable are **different**.

- The **larger** end has a latch and is inserted in the peripheral.
- It should lock into place.
- The colored wires will match the labels.
- To remove it, press the small latch as you pull on the plastic connector.

The connecting ports on the CodeX are also labeled **G** and **S**.



- Turn the CodeX over and look at the **Port Connectors**.
- Connect the smaller end of the cable into a port.
- The Black wire is inserted into the **G** and the Yellow wire into the **S**.

Careful! Never pull on the *wires* - always grasp the plastic connectors when you remove a cable.

Goal:

- Rotate the CodeX in the 3D scene.
  - Click on **PORT3** where you will find the **G** and **S** on the physical CodeX.

Solution:

N/A

Quiz 1 **- Check Your Understanding: Peripherals**

Question 1: Which CodeX Port is the last port on the right?

✓ PORT3

✗ PORT0

✗ PORT1

✗ PORT2

Question 2: What peripheral locks into place when pressed?

✓ Switch

✗ Button

✗ Microswitch

✗ Potentiometer

Question 3: The **black** wire is connected to:

✓ Ground

✗ Signal

✗ Power

✗ Volts

Objective 5 - Connect the Red LED

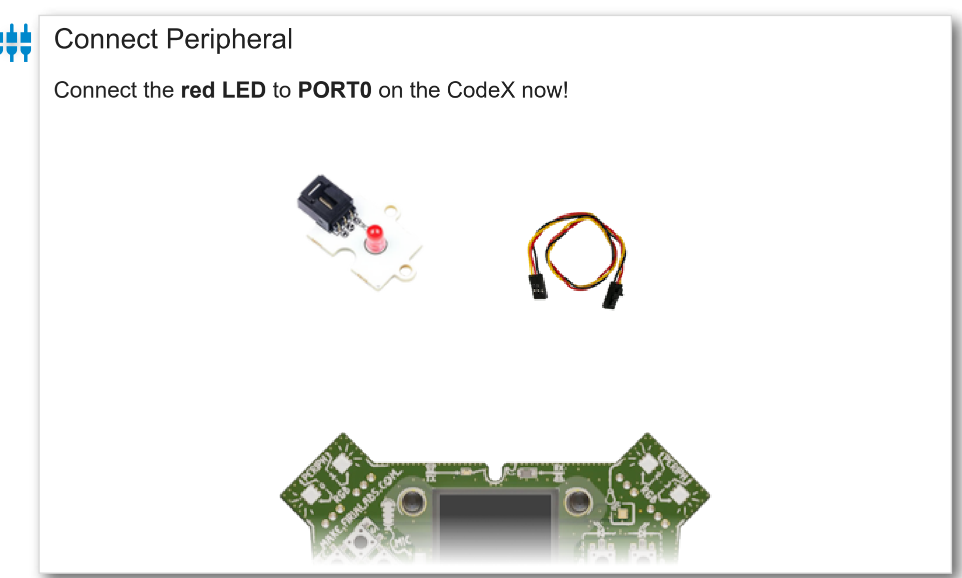
Use the CodeX expansion ports.

The Peripherals course has a new project **symbol** that lets you know it is time to connect a peripheral.

- The Connect Peripheral project **symbol** will contain a peripheral and a connection location on the **CodeX**.
- The connection location will be PORT0, PORT1, PORT2 or PORT3.
- Each PORT is located above the NeoPixel with the same number.

Connect Peripheral

Connect the **red LED** to **PORT0** on the CodeX now!



Next you will write code to turn ON the red LED.

The microcontroller has a pre-loaded library that allows the CodeX to access expansion connectors and peripheral ports.

- You will use the **exp** library to set up peripherals on the CodeX and give them power.
- The code needs to include the following information:
  - The peripheral *port* being used.
  - Type of peripheral: *analog* or *digital*.
  - Purpose of peripheral: *input* or *output*.

The code to set up the red LED looks like this:

```
led = exp.digital_out(exp.PORT0)
```

The code sets up a variable **led** as digital *output*.

Use the `value` property of the variable **led** to turn on and off the LED.

```
led.value = True   # Turn on the LED
led.value = False  # Turn off the LED
```

Create a New File!

Use the **File** → **New File** menu to create a new file called ***PeriphIntro***.

Check the 'Trek!

Follow the CodeTrek to set up the red LED and turn it **ON**.

Run It!

Does the red LED turn **on** and **off**?

CodeTrek:

```
1  from codex import *
2  from time import sleep
```

Import the CodeX library and sleep from the time library.

- This is always the first step for every Lift-Off mission.

```
3
4  # Set up the red LED
5  led = exp.digital_out(exp.PORT0)
```

Set up the red LED on PORT0.

- Tell the CodeX a digital output peripheral is connected on PORT0.

```
6
7  # Main program
8  led.value = True
9  sleep(3)
10 led.value = False
11 sleep(3)
```

Use the `value` property to:

- Turn on the LED (then delay 3 seconds)
- Turn off the LED (then delay 3 seconds)

Hint:

- ## CircuitPython

  The code to set up the peripherals comes from CircuitPython. CircuitPython is a programming language designed to simplify experimenting and learning to code on low-cost microcontroller boards.

  - CircuitPython adds hardware support to the already amazing features of Python.
  - If you already have Python knowledge, you can easily apply that to using CircuitPython.
  - If you have no previous experience, it's really simple to get started.

Goals:

- Create a new file named `PeriphIntro`.
- Set up the **red LED** by assigning the variable `led` as the output of `exp.digital_out(exp.PORT0)`.
- Assign `led.value` as `True`.

Tools Found: Variables

Solution:

```
1  from codex import *
2  from time import sleep #@1
3
4  # Set up the red LED
5  led = exp.digital_out(exp.PORT0)#@2
6
7  # Main program
8  led.value = True
9  sleep(3)
10 led.value = False
11 sleep(3)
```

Objective 6 - Meaningful Code

You are able to turn on and off the LED. But...

Check the 'Trek!

Run It!

Does the red LED turn **on**?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for the peripherals
5  LED_ON = True
6  LED_OFF = False
```

Define constants for the LED.

- They name the purpose of `True` and `False` and make the code more readable.
- You will do this for every mission that uses a red or white LED.

```
7
8  # Set up the red LED
```

```
9  led = exp.digital_out(exp.PORT0)
10
11 # Function that turns on/off the red LED
12 def set_red_led(val):
13     led.value = val
```

Define a function that turns on/off the red LED.

- It takes one parameter `val`.
- Right now, `val` will be either `True` or `False` and is assigned to the LED property `value`.

```
14
15 # Main program
16 while True:
17     set_red_led(LED_ON)
18     sleep(3)
19     set_red_led(LED_OFF)
20     sleep(3)
```

Use an infinite `while True:` loop to turn on and off the red LED.

- Call the `set_red_led()` function.
- Use the constants as arguments.

Hint:

- ## Using Constants with Peripherals

  Why use constants when you could just use `True` and `False`?

  - "Future You" will appreciate the *readability!*
  - `LED_ON` has more meaning than `True`.
  - Circuits do not all operate the same.
  - There are circuits where `False` could turn the LED on!
  - Defining values up front helps avoid errors.

Goals:

- Define constants `LED_ON` and `LED_OFF`.
- Define the function `set_red_led(val)`.
- Assign `val` to `led.value`.
- Call `set_red_led()` with the argument `LED_ON`.

Tools Found: Constants, Functions, Indentation, Keyword and Positional Arguments

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for the peripherals
5  LED_ON = True
6  LED_OFF = False
7
8  # Set up the red LED
9  led = exp.digital_out(exp.PORT0)
10
11 # Function that turns on/off the red LED
12 def set_red_led(val):
13     led.value = val
14
```

```
15 # Main program
16 while True:
17     set_red_led(LED_ON)
18     sleep(3)
19     set_red_led(LED_OFF)
20     sleep(3)
```

Mission 1 Complete

You've completed the first LiftOff Peripherals Kit project!

...and you're at the start of a fantastic **adventure**. From this small first project, your journey will take you to greater heights - more projects are ahead to *challenge* and *amaze* you! A universe of possibilities awaits.

Mission 2 **- Lift-Off!**

This project explores buttons and switches, with a quick refresher on many of the coding concepts you have learned previously!

Mission Briefing:

Your first task in **Mission: Lift Off** is getting the crew's rocket ship off the ground.

The ship is going to need a **power switch**, all personnel will be tuned to a **countdown sequence**, and mission control needs a **launch button** to fire the engines and blast the crew's rocket out of the atmosphere.

Project Goals:

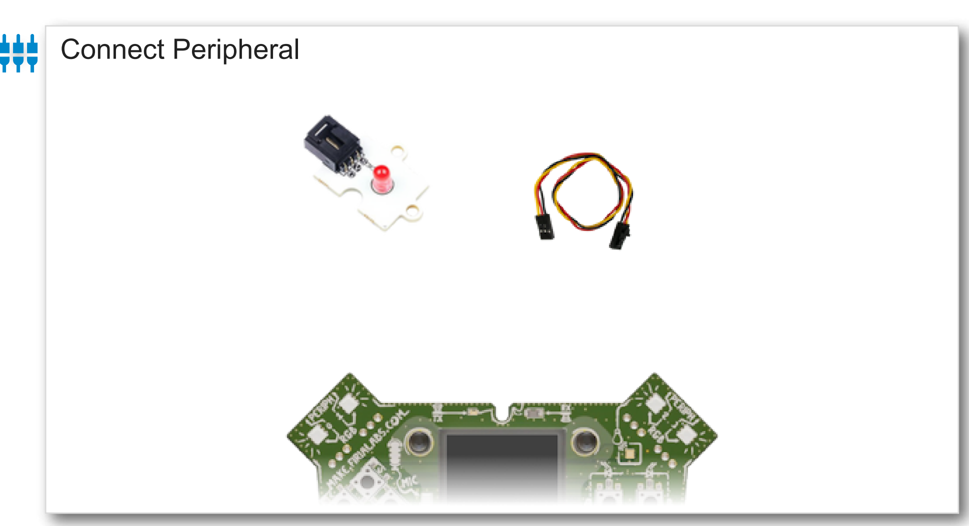
- Utilize buttons, switches, and LEDs together
- Understand the differences between a button and a switch
- Refresh your **Python** knowledge
- Get the crew's mission to Mars underway!!

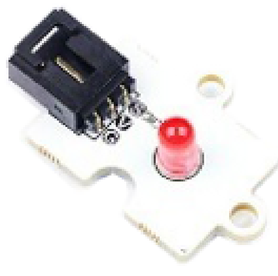
Ready to get started?

Objective 1 **- Add a Switch**

The first step is getting power on the ship!

Connect Peripheral



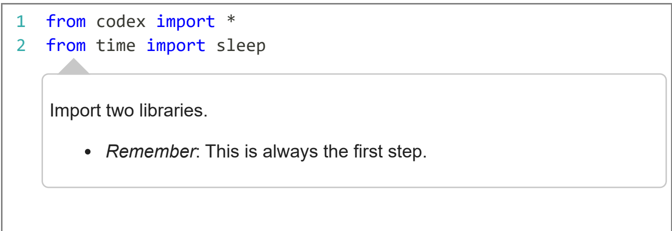


Create a New File!

Use the **File** → **New File** menu to create a new file called ***LiftOff***.

Check the 'Trek!

CodeTrek:



```
1  from codex import *
2  from time import sleep
```

Import two libraries.

- *Remember*: This is always the first step.

```
3
4  # Constants for LED light
5  # TODO: Define constants for LED_ON and LED_OFF
```

The LED_ON is True and LED_OFF is False.

```
6
7  # Set up the LED
8  red_led = exp.digital_out(exp.PORT0)
```

Set up the red LED on PORT0.

- Tell the CodeX a digital output peripheral is connected on PORT0.

```
9
10 def set_red_led(val):
11     red_led.value = val
```

Define a function to turn the LED on or off.

```
12
13 set_red_led(LED_ON)
14 sleep(1)
15 set_red_led(LED_OFF)
```

Make sure the LED is properly connected by using this test code.

- Remember to import sleep from the time module.

Hints:

- ## Look familiar?

  This code is very similar to the program for Mission 1. You can open **PeriphIntro** and copy and paste.

- Use a delay (sleep) in between turning the light on and turning it off.
  - This is for testing.

Goals:

- Create a new file named `LiftOff`.
- Define the constants `LED_ON` and `LED_OFF`.
- Define the function `set_red_led(val)`.
- Assign `val` to `red_led.value`.
- Call `set_red_led()` twice to turn on and off the red light

Tools Found: Constants, Functions

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for LED light
5  LED_ON = True
6  LED_OFF = False
7
8  # Set up the LED
9  red_led = exp.digital_out(exp.PORT0)
```

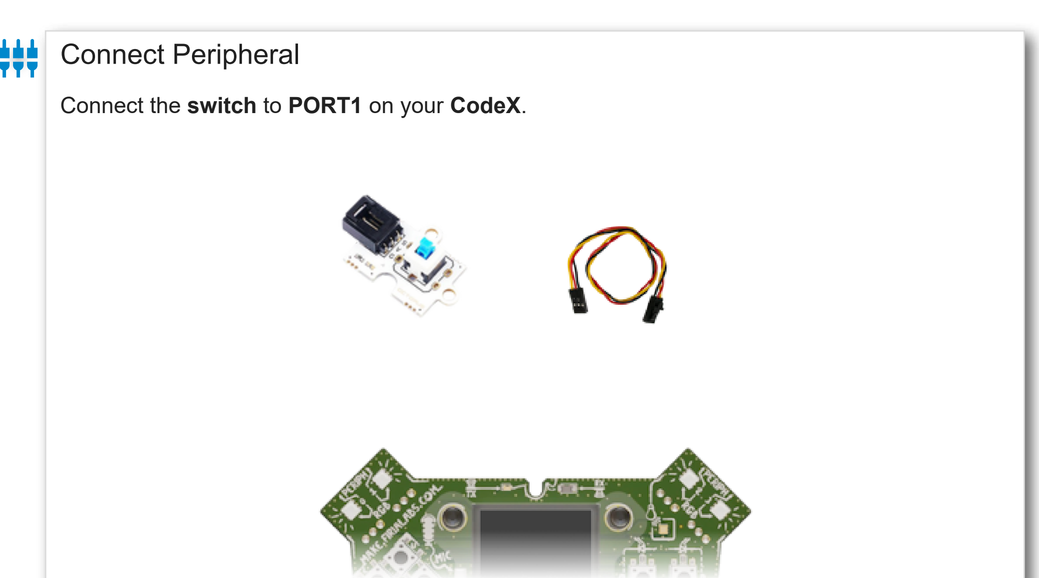
```

def set_red_led(val):
    red_led.value = val

set_red_led(LED_ON)
sleep(1)
set_red_led(LED_OFF)


```

Objective 2 **- Turn the Power ON!!**

You will need a switch to turn on power for the shuttle's launch operation.

A switch is a 🔧peripheral that can be set to a 🔧binary **ON** or **OFF** position.

- The switch in your 🔧LiftOff Kit is similar to a light switch in your **home**.



Connect Peripheral

Connect the **switch** to **PORT1** on your **CodeX**.

Physical Interaction: *Try it!*

Press the switch a few times to see how it moves!

Now that your switch is connected you can use it to turn the power ON for the launch operation.

The switch is a digital **input** peripheral, so use the `exp.digital_in()` 🔧function to set it up.

- After it is set up, `switch.value` will return the position of your switch as either `True` or `False`.

```
switch = exp.digital_in(exp.PORT1)
```

You want to check the switch's position continuously. *How can you make the program keep checking?*

If you reach back into your **Python** knowledge, you might remember that 🔧loops allow a program to run sections of code multiple times.

- With a 🔧loop, you can update the position of your switch in real time.

Check the 'Trek!

Follow the CodeTrek to set up the switch and use a `while True:` loop to read the switch's position.

- Use `switch.value` as the argument in the `set_red_led()` function call.

Test the switch for both positions.

Physical Interaction: *Try it*

Run the code. While it is running:

- Extend the switch to the **OUT** position. Observe the switch value and the LED.
- Push the switch to the **IN** position. Observe the switch value and the LED.

Did you notice that the red LED turned **ON** when the switch was **OUT** and turned **OFF** when the switch was **IN**?

If you are thinking, *"Hmmm... the power should come **ON** when the switch is **IN**"* ... You will get to that soon, but it is **important** to realize that you can **choose** which way it works in your own projects

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for LED light
5  LED_ON = True
6  LED_OFF = False
7
8  # Set up the LED
9  red_led = exp.digital_out(exp.PORT0)
10
11 # Set up the switch
12 switch = exp.digital_in(exp.PORT1)
```

Set up the switch on PORT1.

- Tell the CodeX a digital input peripheral is connected on PORT1.

```
13
14 def set_red_led(val):
15     red_led.value = val
16
17 # Main program
18 while True:
19     display.print(switch.value)
20     set_red_led(switch.value)
```

Use `switch.value` as the argument to the `set_red_led()` function call.

- Remember: `switch.value` is either `True` or `False`, just like the constants `LED_ON` and `LED_OFF`.
- For testing, add `display.print(switch.value)` to see what value is returned.
- The value will be printed on the CodeX display.
- Use a `while True:` loop to read the switch continuously.

```
21
```

Hint:

- ## Up and Down

  Adding a display.print() statement to see the value returned by the switch's position on the CodeX display is a valuable tool for testing.

  - You can observe the value of the peripheral.
  - AND you can observe the result of the value.

Goals:

- Set up the **switch** by assigning the variable `switch` as the output of `exp.digital_in(exp.PORT1)`.
- Inside a `while True:` loop:

- Use `switch.value` as the 🔧argument in the 🔧function call to `set_red_led()`.

Tools Found: CPU and Peripherals, Binary Numbers, LiftOff Kit, Functions, Loops, Variables, Keyword and Positional Arguments

Solution:

```
1  from codex import *
2  from time import sleep
3
4  LED_ON = True
5  LED_OFF = False
6
7  # Set up the LED
8  led = exp.digital_out(exp.PORT0)
9
10 # Set up the Switch
11 switch = exp.digital_in(exp.PORT1)
12
13 def set_red_led(val):
14     led.value = val
15
16 # Main program
17 while True:
18     display.print(switch.value)
19     set_red_led(switch.value)
20
21
```

Objective 3 - Clean Energy

Now, how do you make the LED turn ON when the switch is pressed *IN?*

Start by mapping it out. Say you want:

- **ON** if the switch is pressed **IN**
- **OFF** if the switch is extended **OUT**

Since `switch.value` returns `False` when pressed **IN**, declare that `False` means `POWER_ON`.

Define the 🔧constant near the top of your file:

```
POWER_ON = False
```

*"Now, how can I turn the LED **on** `if` the switch value is `POWER_ON`?"*

Get your **Python** thinking cap on again. Think... Think... That's it!!

You can use 🔧branching to turn on and off your LED!

An `if` statement allows your code to choose its path based on a 🔧condition.

- If `switch.value` is `POWER_ON`, or `False`, turn the LED **ON**.
- Otherwise, turn the LED **OFF**.

Check the 'Trek!

Follow the CodeTrek to add an `if` statement to the loop to turn on/off the LED.

Physical Interaction: *Try it*

Run your code. Press the switch **IN** to turn **ON** the red LED, and **OUT** to turn **OFF** the red LED.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants
5  LED_ON = True
6  LED_OFF = False
7
8  POWER_ON = False   # Switch
```

Define a constant to indicate **ON** when the switch is pushed **IN**.

- Constants are defined near the top of the file.

```
9
10 # Set up the LED
11 red_led = exp.digital_out(exp.PORT0)
12
13 # Set up the switch
14 switch = exp.digital_in(exp.PORT1)
15
16 def set_red_led(val):
17     red_led.value = val
18
19 while True:
20     if switch.value == POWER_ON:
```

The `==` operator checks if the current `switch.value` is the same as `POWER_ON` and returns a Boolean value.

```
21         set_red_led(LED_ON)
22     else:
23         set_red_led(LED_OFF)
```

If `True` use `LED_ON` as the argument for turning on the light.
Otherwise use `LED_OFF` as the argument for turning off the light.

Hints:

- ## Compare, not assign

  Watch out for the *double equals* `==` comparison operator!

  - It checks if two things are the **same** (or equal).
  - The single *equals* `=` assigns a value and is **NOT** used for comparing.

- ## ON and OFF

  The switch's value will be used for branching. Then use the constants `LED_ON` and `LED_OFF` as the arguments to the function call `set_red_led(val)` instead of `switch.value`.

Goals:

- Define a constant `POWER_ON` with the value `False`.
- Use an `if` statement with the condition `switch.value == POWER_ON`.
- When `True`, call `set_red_led(LED_ON)`.

Tools Found: Constants, Branching, bool

Solution:

```
1  from codex import *
2  from time import sleep
```

```
3
4  # Constants
5  LED_ON = True
6  LED_OFF = False
7
8  POWER_ON = False   # Switch
9
10 # Set up the LED
11 red_led = exp.digital_out(exp.PORT0)
12
13 # Set up the switch
14 switch = exp.digital_in(exp.PORT1)
15
16 def set_red_led(val):
17     red_led.value = val
18
19 while True:
20     if switch.value == POWER_ON:
21         set_red_led(LED_ON)
22     else:
23         set_red_led(LED_OFF)
24
```

Objective 4 **- Button vs Switch**

The switch gives your launch power, but you are going to need a button to start the countdown to lift-off!

"What is the difference between a button and a switch?"

- As strange as it seems, the **CodeX** does not see a difference!
- Both the switch and the button are **digital input** peripherals.
- A button and a switch are both read with `exp.digital_in()`.

The difference between a button and a switch is entirely **mechanical**:

- A switch locks in the **ON** position.
- A button only stays in the **ON** position while it is pressed.

You will keep the **switch** for the power and add the **button** for the countdown.

Connect Peripheral

Connect the **button** to **PORT2** on your CodeX.

Add code to control the red light with your button!

Just like the switch, the button returns `True` when it is **up** (not pressed) and `False` when it is **down** (pressed).

- You will want to **declare** that `False` means `PRESSED`.
- You can define a constant `PRESSED` and assign it the value `False`, like you did for the switch.

Then for testing:

- Change the 🔧branch statement in your 🔧loop to compare `button.value` to `PRESSED`.

Check the 'Trek!

Follow the CodeTrek to set up the button and use it to control the red LED.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants
5  LED_ON = True
6  LED_OFF = False
7
8  POWER_ON = False   # Switch
9  PRESSED = False    # Button
```

Define another constant to indicate **ON** when the button is pushed.

```
10
11 # Set up the LED
12 red_led = exp.digital_out(exp.PORT0)
13
14 # Set up the switch
15 switch = exp.digital_in(exp.PORT1)
16
17 # Set up the button
18 button = exp.digital_in(exp.PORT2)
```

Set up the button on PORT2.

- Tell the CodeX a digital input peripheral is connected on PORT2.

```
19
20 def set_red_led(val):
21     red_led.value = val
22
23 # Main program
24 while True:
25     if button.value == PRESSED:
```

Use the button to control the red LED.

- Replace the switch's value with the button's value in the condition.
- Replace `POWER_ON` with `PRESSED`.

```
26         set_red_led(LED_ON)
27     else:
28         set_red_led(LED_OFF)
```

Hint:

- ## The Need for Constants

  When the button is pressed, it's value is `False`. You can define a constant for `False`, like you did for the switch, and use it in a condition.

  Then you can turn **on** the light when the button is pressed, or `False`, and turn **off** the light when the button is not pressed, or `True`.

Goals:

- Set up the **button** by assigning the 🔧variable `button` as the output of `exp.digital_in(exp.PORT2)`.

- Define the constant `PRESSED`.
- Use an `if` statement with the condition `button.value == PRESSED`.
- When `True`, call `set_red_led(LED_ON)`.

Tools Found: Constants, Branching, Loops, Variables, bool

Solution:

```
1  from codex import *
2  from time import sleep
3
4  LED_ON = True
5  LED_OFF = False
6
7  POWER_ON = False   # Switch
8  PRESSED = False    # Button
9
10 # Set up the LED
11 red_led = exp.digital_out(exp.PORT0)
12
13 # Set up the switch
14 switch = exp.digital_in(exp.PORT1)
15
16 # Set up the button
17 button = exp.digital_in(exp.PORT2)
18
19 def set_red_led(val):
20     red_led.value = val
21
22 while True:
23     if button.value == PRESSED:
24         set_red_led(LED_ON)
25     else:
26         set_red_led(LED_OFF)
```

Quiz 1 - Check Your Understanding: Button vs Switch

Question 1: What type of peripheral is the red LED?

✓ Digital Output

✗ Digital Input

✗ Analog Input

✗ Analog Output

Question 2: What type of peripheral is the switch?

✓ Digital Input

✗ Digital Output

✗ Analog Input

✗ Analog Output

Question 3: What value is assigned to `POWER_ON`?

✓ `False`

✗ `True`

`0`

`1`

Question 4: Which peripheral momentarily stays ***on*** when you press it but does not lock into place?

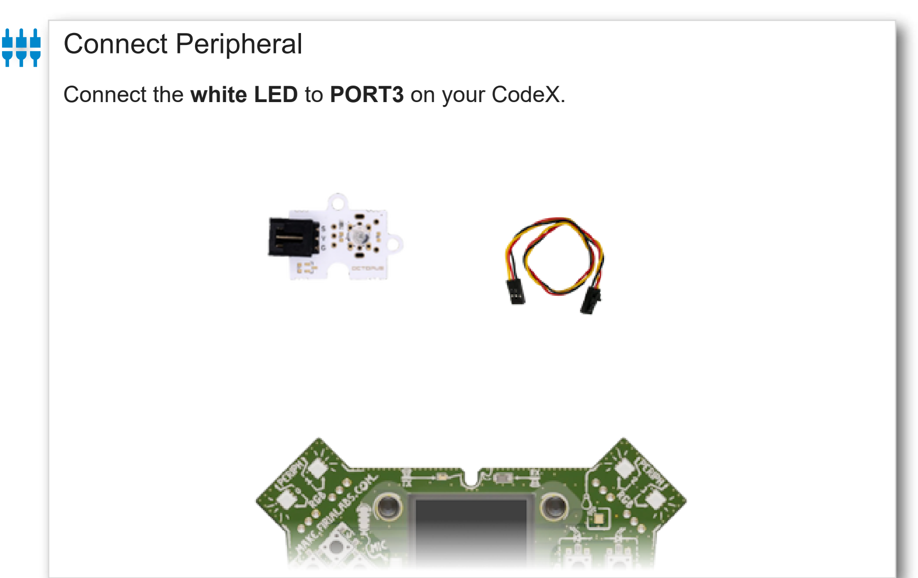
- ✗ Red LED
- ✓ Button
- ✗ Switch

***Objective 5* - Countdown!**

A ***flashing*** *white LED* will make a great **Countdown Indicator**.

Connect Peripheral

Connect the **white LED** to **PORT3** on your CodeX.



Here is the **algorithm** to create a countdown when the button is pressed:

- Add code to set up the white LED, similar to the red LED.
- Define a new function `def set_white_led()` that turns on/off the white LED.
  - The function should set the **white LED** to the parameter **val**, similar to the `set_red_led()` function.
- Define a new `def countdown()` function so the launch operation can prepare for lift-off!
  - The function will flash the white light, display numbers and use delays to make a launch operation.

This Objective requires two more functions, which makes your code meaningful. That's because a function is a form of procedural abstraction.

Concept: *Abstraction*

Your code will get more **complicated** and harder to read as you add more peripherals to a project. It is **crucial** to stay **organized**!

A powerful tool for organizing code is **abstraction**. When you define a function like `set_white_led()` or `countdown()` you are creating a new *abstraction!*

- Now the rest of the code doesn't have to know about the details of what the function does or how it does it.
- In the future you could completely change how the *LED* turns on/off or how the *countdown* looks, and all the other code would keep working without change!

Check the 'Trek!

Follow the CodeTrek to define two functions:

- `set_white_led()` for turning on/off the white light
- `countdown()` for flashing the white light to prepare for launch.

Then call `countdown()` in the `if` statement when the button is pressed.

Physical Interaction: *Try it!*

Run the code. Press the button to start the countdown sequence, and then end the program.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants
5  LED_ON = True
6  LED_OFF = False
7
8  POWER_ON = False   # Switch
9  PRESSED = False    # Button
10
11 # Set up the LED
12 red_led = exp.digital_out(exp.PORT0)
13 white_led = exp.digital_out(exp.PORT3)
```

Set up the white LED on PORT3.

```
14
15 # Set up the switch
16 switch = exp.digital_in(exp.PORT1)
17
18 # Set up the button
19 button = exp.digital_in(exp.PORT2)
20
21 def set_red_led(val):
22     red_led.value = val
23
24 def set_white_led(val):
25     # TODO: set the white LED to `val`
```

Define the function `set_white_led()` to turn on/off the white LED.

- Set the value of the white LED to the parameter `val`

```
26
27 # Lift off countdown when button is pressed
28 def countdown():
29     display.print("3", scale=9)
30     set_white_led(LED_ON)
31     sleep(0.5)
32     set_white_led(LED_OFF)
33     sleep(0.5)
34     display.print("2", scale=9)
35     set_white_led(LED_ON)
36     sleep(0.5)
37     set_white_led(LED_OFF)
38     sleep(0.5)
39     display.print("1", scale=9)
40     set_white_led(LED_ON)
41     sleep(0.5)
```

Define the `countdown()` function to countdown the launch operation when the button is pressed.

- Use `display.print()` to show the countdown number.
- Flash the white light on and off for each number.

- Use a half-second delay between each cycle of number and flashing white light.

```
42
43  # Main program
44  while True:
45      if button.value == PRESSED:
46          countdown()
47          break
```

Modify the `if` statement to call `countdown()` when the button is pressed.

- Then `break` to end the program.

```
48      else:
49          set_red_led(LED_OFF)
50
51  # Program ends - turn off LED
52  set_white_led(LED_OFF)
```

Turn off the white LED after the `while True:` loop ends.

- This last command is needed because the white LED will stay on, even after the program ends.
- Watch your indenting! The last statement is **NOT** inside the `while True:` loop.

Hints:

- ## Red and White

  The code for setting up the white LED and turning it on/off with a function is very similar to the code for the red LED.

  You can even copy and paste the code and then make changes from **red** to **white**.

- ## Delay

  If you remember, a *delay* is just a pause in your program.

  - It is as simple as calling the `sleep()` function with a *half-second* argument.

Goals:

- Define the function `set_white_led(val)`.
- Define the function `countdown()`.
- Flash the **white LED** three times.
- Use an `if` statement with the condition `button.value == PRESSED`.
- When `True`, call `countdown()`.

Tools Found: Functions, Parameters, Arguments, and Returns, Timing, CPU and Peripherals, bool

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for LED light
5  LED_ON = True
6  LED_OFF = False
7
8  POWER_ON = False   # Switch
9  PRESSED = False    # Button
10
11 # Set up the LED
12 red_led = exp.digital_out(exp.PORT0)
13 white_led = exp.digital_out(exp.PORT3)
```

```
14
15  # Set up the switch
16  switch = exp.digital_in(exp.PORT1)
17
18  # Set up the button
19  button = exp.digital_in(exp.PORT2)
20
21  def set_red_led(val):
22      red_led.value = val
23
24  def set_white_led(val):
25      white_led.value = val
26
27  # Lift off countdown when button is pressed
28  def countdown():
29      display.print("3", scale=9)
30      set_white_led(LED_ON)
31      sleep(0.5)
32      set_white_led(LED_OFF)
33      sleep(0.5)
34      display.print("2", scale=9)
35      set_white_led(LED_ON)
36      sleep(0.5)
37      set_white_led(LED_OFF)
38      sleep(0.5)
39      display.print("1", scale=9)
40      set_white_led(LED_ON)
41      sleep(0.5)
42
43  # Main program
44  while True:
45      if button.value == PRESSED:
46          countdown()
47          break
48      else:
49          set_red_led(LED_OFF)
50
51  # Program ends - turn off LED
52  set_white_led(LED_OFF)
```

Objective 6 - Switches and Buttons... Oh My!

What if you only want your *countdown sequence* to happen if the *power is ON?*

The button controls the countdown, but the switch controls the power.

Concept

There is **always** more than one way to accomplish something in code!

This problem is no different. There are **MANY** ways to solve this.

Check the 'Trek!

CodeTrek:

```
1   from codex import *
2   from time import sleep
3
4   # Constants
5   LED_ON = True
6   LED_OFF = False
7
8   POWER_ON = False    # Switch
9   PRESSED = False     # Button
10
11  # Set up the LED
12  red_led = exp.digital_out(exp.PORT0)
13  white_led = exp.digital_out(exp.PORT3)
```

```
14
15  # Set up the switch
16  switch = exp.digital_in(exp.PORT1)
17
18  # Set up the button
19  button = exp.digital_in(exp.PORT2)
20
21  def set_red_led(val):
22      red_led.value = val
23
24  def set_white_led(val):
25      white_led.value = val
26
27  # Lift off countdown when button is pressed
28  def countdown():
29      display.print("3", scale=9)
30      set_white_led(LED_ON)
31      sleep(0.5)
32      set_white_led(LED_OFF)
33      sleep(0.5)
34      display.print("2", scale=9)
35      set_white_led(LED_ON)
36      sleep(0.5)
37      set_white_led(LED_OFF)
38      sleep(0.5)
39      display.print("1", scale=9)
40      set_white_led(LED_ON)
41      sleep(0.5)
42
43  # Main program
44  while True:
45      if button.value == PRESSED and switch.value == POWER_ON:
46          countdown()
47          break
```

The first condition checks if the button is pressed **and** the switch is on.

- If so, call `countdown()`.

```
48      elif switch.value == POWER_ON:
49          set_red_led(LED_ON)
```

The second condition checks if the switch is on.

- If so, turn **on** the **red** LED.
- You don't need to check if the button isn't pressed.

```
50      else:
51          set_red_led(LED_OFF)
```

If neither condition is `True` then turn **off** the red LED.

- The button isn't pressed and the switch is off.

```
52
53  # Program ends - turn off LED
54  set_white_led(LED_OFF)
55  set_red_led(LED_OFF)
```

After the `while True:` loop ends, turn off the red LED.

- You already turn off the white LED. Include the red LED as well.

Hint:

- ## Order!

  When you have more than two conditions, the order of the conditions makes a difference.

The CodeX starts checking with the first condition and stops checking the first time a condition is `True`. Any remaining conditions will NOT be checked.

Make sure you put the most restrictive condition first, and go in order to get the results you want.

Goals:

- Modify the `if` statement to use **three** conditions.
- `if`, `elif`, and `else`!
- Use an `if` statement with the condition `button.value == PRESSED and switch.value == POWER_ON`.
- When `True`, call `countdown()`.

Tools Found: bool, Branching, Logical Operators, undefined

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants
5  LED_ON = True
6  LED_OFF = False
7
8  POWER_ON = False   # Switch
9  PRESSED = False    # Button
10
11 # Set up the LED
12 red_led = exp.digital_out(exp.PORT0)
13 white_led = exp.digital_out(exp.PORT3)
14
15 # Set up the switch
16 switch = exp.digital_in(exp.PORT1)
17
18 # Set up the button
19 button = exp.digital_in(exp.PORT2)
20
21 def set_red_led(val):
22     red_led.value = val
23
24 def set_white_led(val):
25     white_led.value = val
26
27 # Lift off countdown when button is pressed
28 def countdown():
29     display.print("3", scale=9)
30     set_white_led(LED_ON)
31     sleep(0.5)
32     set_white_led(LED_OFF)
33     sleep(0.5)
34     display.print("2", scale=9)
35     set_white_led(LED_ON)
36     sleep(0.5)
37     set_white_led(LED_OFF)
38     sleep(0.5)
39     display.print("1", scale=9)
40     set_white_led(LED_ON)
41     sleep(0.5)
42
43 # Main program
44 while True:
45     if button.value == PRESSED and switch.value == POWER_ON:
46         countdown()
47         break
48     elif switch.value == POWER_ON:
49         set_red_led(LED_ON)
50     else:
51         set_red_led(LED_OFF)
52
```

```
53  # Program ends - turn off LED
54  set_white_led(LED_OFF)
55  set_red_led(LED_OFF)
```

Quiz 2 - Check Your Understanding: Coding Concepts

Question 1: What statement allows you to add conditional branches to an `if` statement?

- ✓ `elif`
- ✗ `break`
- ✗ `and`
- ✗ `else`

Question 2: What statement is executed when no other condition is `True`?

- ✓ `else`
- ✗ `elif`
- ✗ `break`
- ✗ `and`

Question 3: What keyword is a logical operator?

- ✓ `and`
- ✗ `elif`
- ✗ `else`
- ✗ `break`

Question 4: What statement exits a loop immediately when called?

- ✓ `break`
- ✗ `elif`
- ✗ `and`
- ✗ `else`

Objective 7 - Lift-Off

Are you ready for launch?

Simulate a lift-off on the **CodeX** using `display.fill()` and colors that simulate a lift-off.

You can use built-in colors like this:

```
display.fill(RED)
sleep(0.25)
display.fill(ORANGE)
sleep(0.25)
display.fill(YELLOW)
sleep(0.25)
```

```
display.fill(WHITE)
sleep(0.25)
```

Define a function for `lift_off()`. You will call this function after `countdown()` and before ending the program.

Check the 'Trek!

Follow the CodeTrek to define the function `lift_off()` and call it after `countdown()`.

And now you are ready to send this ship into outer space!!

Physical Interaction: *Houston, we have lift-off!*

Press the switch and then the button to launch the rocket!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants
5  LED_ON = True
6  LED_OFF = False
7
8  POWER_ON = False   # Switch
9  PRESSED = False    # Button
10
11 # Set up the LED
12 red_led = exp.digital_out(exp.PORT0)
13 white_led = exp.digital_out(exp.PORT3)
14
15 # Set up the switch
16 switch = exp.digital_in(exp.PORT1)
17
18 # Set up the button
19 button = exp.digital_in(exp.PORT2)
20
21 def set_red_led(val):
22     red_led.value = val
23
24 def set_white_led(val):
25     white_led.value = val
26
27 # Lift off countdown when button is pressed
28 def countdown():
29     set_white_led(LED_ON)
30     sleep(0.5)
31     set_white_led(LED_OFF)
32     sleep(0.5)
33     set_white_led(LED_ON)
34     sleep(0.5)
35     set_white_led(LED_OFF)
36     sleep(0.5)
37     set_white_led(LED_ON)
38
39 # Final blast-off after countdown
40 def lift_off():
41     display.clear()
42     display.fill(RED)
43     sleep(0.25)
44     display.fill(ORANGE)
45     sleep(0.25)
46     display.fill(YELLOW)
47     sleep(0.25)
48     display.fill(WHITE)
49     sleep(0.25)
```

Your color sequence can look like this.

- Use the colors you want and as many colors as you want.
- Also, choose the delay time between colors.

```
50      display.clear()
51      display.print("LIFT-OFF", scale=4)
52      sleep(1)
```

Optional!

You can display a message before, during, or after the color sequence.

```
53
54  while True:
55      if button.value == PRESSED and switch.value == POWER_ON:
56          countdown()
57          # TODO: Call the lift_off() function
```

Call the `lift_off()` function before ending the program.

```
58          break
59      elif switch_val == POWER_ON:
60          set_red_led(LED_ON)
61      else:
62          set_red_led(LED_OFF)
63
64  # Program ends - turn off LED
65  set_white_led(LED_OFF)
66  set_red_led(LED_OFF)
```

Hints:

- ## RGB Colors

  If you used RGB to define colors in other missions, you can use them here to display shades of colors, like this.

  Here is an example of shades of red:

```
display.fill((203, 11, 28))
sleep(0.25)
display.fill((169, 10, 24))
sleep(0.25)
display.fill((131, 9, 19))
sleep(0.25)
display.fill((92, 7, 14))
sleep(0.25)
```

- ## Show and Tell

  For an added touch, you can include a message with your launch.

  For example, you can use `display.print("Lift-Off!")` somewhere in your launch sequence.

  You can also add extra features to your lift-off, like pixels flashing or an audio file, to make your lift-off unique to your program.

Goals:

- Define the function `lift_off()`.
- Call the following functions.
  - `display.fill(RED)`
  - `display.fill(ORANGE)`
  - `display.fill(YELLOW)`
  - `display.fill(WHITE)`
- Call `lift_off()`.

Tools Found: Functions

Solution:

```
from codex import *
from time import sleep

# Constants
LED_ON = True
LED_OFF = False

POWER_ON = False   # Switch
PRESSED = False    # Button

# Set up the LED
red_led = exp.digital_out(exp.PORT0)
white_led = exp.digital_out(exp.PORT3)

# Set up the switch
switch = exp.digital_in(exp.PORT1)

# Set up the button
button = exp.digital_in(exp.PORT2)

# Turn on/off the LEDs
def set_red_led(val):
    red_led.value = val

def set_white_led(val):
    white_led.value = val

# Countdown sequence with flashing white LED when button is pressed
def countdown():
    display.print("3", scale=9)
    set_white_led(LED_ON)
    sleep(0.5)
    set_white_led(LED_OFF)
    sleep(0.5)
    display.print("2", scale=9)
    set_white_led(LED_ON)
    sleep(0.5)
    set_white_led(LED_OFF)
    sleep(0.5)
    display.print("1", scale=9)
    set_white_led(LED_ON)
    sleep(0.5)

# Final lift-off sequence (can be different for each student)
def lift_off():
    display.clear()
    display.fill(RED)
    sleep(0.25)
    display.fill(ORANGE)
    sleep(0.25)
    display.fill(YELLOW)
    sleep(0.25)
    display.fill(WHITE)
    sleep(0.25)
    display.clear()
    display.print("LIFT-OFF", scale=4)
    sleep(1)

# === MAIN PROGRAM ===
while True:
    if button.value == PRESSED and switch.value == POWER_ON:
        countdown()
        lift_off()
        break
    elif switch.value == POWER_ON:
        set_red_led(LED_ON)
    else:
        set_red_led(LED_OFF)
```

```
69
70  # Program ends - turn off LED
71  set_white_led(LED_OFF)
72  set_red_led(LED_OFF)
```

Mission 2 Complete

You've completed project Lift-Off!

...and you helped launch the crew into outer space. What will they need now that they have left Earth's amosphere?

Mission 3 - Conserve Energy!

This project introduces an analog and digital sensor and shows how to dim an LED with software!

Mission Briefing:

The crew's rocket ship is hurtling through space. Energy conservation will be **critical** to surviving the long journey to **Mars**.

Project: Conserve Energy will guide you through reducing power consumption in the ship's lighting system.

You will need to *detect motion* in the crew compartment to turn **on** the lights and *reduce power* by dimming them down when no motion is detected.

Project Goals:

- Take an analog input from a **potentiometer** (knob)
- Learn software techniques to dim an LED
- Use a **digital sensor** (the PIR Motion Sensor)
- Conserve critical energy for mission success!!

Objective 1 - Lighting System

The first step is constructing a lighting system for the shuttle.

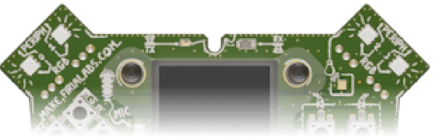
Add the ship's main lights!

- The white LED will be the lighting system for the shuttle's main room

Connect Peripheral

Connect the **white LED** to **PORT0** on your CodeX.

- Disconnect all other peripherals.



Create a New File!

Use the **File → New File** menu to create a new file called ***ConserveEnergy***.

Check the 'Trek!

Follow the CodeTrek to set up the white LED and turn it **on**.

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep
```

Remember to import the libraries you need.

```
3
4  # Constants for peripherals
5  # TODO: define constants for LED_ON and LED_OFF
```

Define your constants for the LED.

- `LED_ON` is `True`
- `LED_OFF` is `False`

```
6
7  # Set up the peripherals
8  led = exp.digital_out(exp.PORT0)
```

Set up the white LED on PORT0

```
9
10 def set_led(val):
11     led.value = val
12
13 # Main program
14 while True:
15     set_led(LED_ON)
```

Define the `set_led()` function.

- Call the function in the `while True` loop.
- Turn the LED **ON** by using `LED_ON` as the argument.

```
16
```

Hints:

- ## Look familiar?

  This code is very similar to code used at the beginning of Mission 2. You can open the file and copy and paste the part you need.

- ## LED setup

  In previous missions, you designated variables for `red_led` and `white_led`. For this mission, you will only use one LED, so the variable name can be simplified.

Goals:

- Create a new file named `ConserveEnergy`.
- Define the function `set_led(val)`.
- Call `set_led()`.
- Use `LED_ON` as the argument to turn the LED **ON**.

Tools Found: Functions

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = True
6  LED_OFF = False
7
8  # Set up the peripherals
9  led = exp.digital_out(exp.PORT0)
10
11 def set_led(val):
12     led.value = val
13
14 # Main program
15 while True:
16     set_led(LED_ON)
17
```

Objective 2 **- Power Cycle the Lights?**

Caution

Concept: *Fast Timers*

You've been using the `sleep()` function from the `import time` library module.

- There are even *more* timer features available in Python's **time module**!

For example:

`sleep_ms()` is a delay just like `sleep()` but with *milliseconds (ms)* instead of *seconds*.

- 1000 milliseconds (ms) = 1 seconds

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep_ms
```

Change the `from time import sleep` statement to use milliseconds.

```
3
4  # Constants for peripherals
5  LED_ON = True
6  LED_OFF = False
7
8  # Set up the peripherals
9  led = exp.digital_out(exp.PORT0)
10
11 def set_led(val):
12     led.value = val
13
14 # Main program
15 while True:
16     set_led(LED_ON)
17     sleep_ms(0.25)
18     set_led(LED_OFF)
19     sleep_ms(0.75)
```

Use function calls to `sleep_ms()` to dim the light.

- Use `0.25` as the argument for `LED_ON`
- Use `0.75` as the argument for `LED_OFF`

Hints:

- ## Fast Timers

  Some functions, like `sleep_ms()` are not available in regular Python, but they are available in some microcontroller versions, like CircuitPython and MicroPython.

- ## Here are some fun facts:

  - In human vision there is a *"Critical Flicker-Fusion Frequency"*
  - ...anything flashing faster than *that* will not be perceived as flickering!
  - Your 🔧loop cycles every 1000us $\rightarrow freq = \frac{1}{0.001 sec} = 1000\ Hz$
  - That's well above the *Critical Flicker-Fusion Frequency!*
  - **AND** you just **reduced** energy consumption by 75%!!
  - The LED appears **dimmer** than it was because there is less total energy put into the LED.

- ## Try your skills

  Try different arguments for `sleep_ms()` to see how the light changes.

  - Experiment to find the *"Critical Flicker-Fusion Frequency"!*

Goals:

- 🔧Import `sleep_ms`.
- In a `while True` 🔧loop:
  1. Turn the LED **ON**.
  2. 🔧Sleep for 0.25ms.
  3. Turn the LED **OFF**.
  4. Sleep for 0.75ms.

Tools Found: Time Module, Timing, import, Loops

Solution:

```
1  from codex import *
2  from time import sleep_ms
3
4  # Constants for peripherals
5  LED_ON = True
6  LED_OFF = False
7
8  # Set up the peripherals
9  led = exp.digital_out(exp.PORT0)
10
11 def set_led(val):
12     led.value = val
13
14 # Main program
15 while True:
16     set_led(LED_ON)
17     sleep_ms(0.25)
18     set_led(LED_OFF)
19     sleep_ms(0.75)
```

***Quiz 1* - Check for Understanding: Time**

Question 1: Which function will delay code using milliseconds?

- ✓ `sleep_ms()`
- ✗ `sleep()`
- ✗ `sleep_us()`
- ✗ `delay_ms()`

Question 2: How many milliseconds are in **one** second?

- ✓ 1,000
- ✗ 100
- ✗ 10,000
- ✗ 1,000,000

Objective 3 - PWM (Pulse-width Modulation)

It could get complicated to add mulitple `sleep_ms()` calls in your loop.

Is there a way to make the ON and OFF happen automatically?

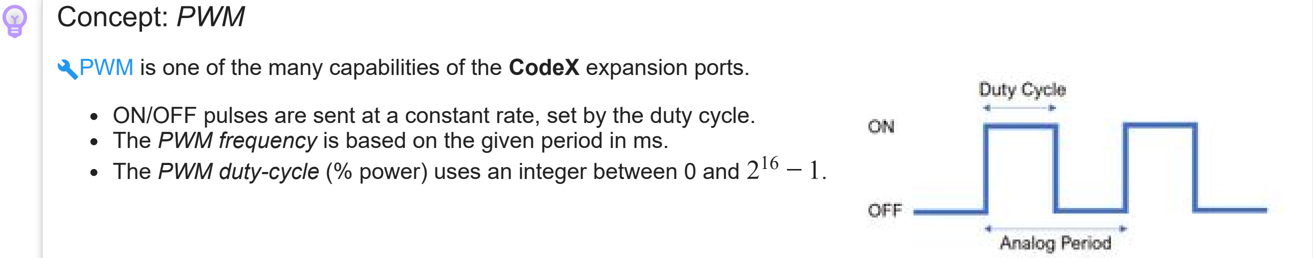
YES!! You can use Pulse-width Modulation (PWM).

- PWM allows you to turn the power **ON** and **OFF** behind the scenes at a **fast rate** by just setting a few values.

Concept: *PWM*

PWM is one of the many capabilities of the **CodeX** expansion ports.

- ON/OFF pulses are sent at a constant rate, set by the duty cycle.
- The *PWM frequency* is based on the given period in ms.
- The *PWM duty-cycle* (% power) uses an integer between 0 and $2^{16} - 1$.



Check the 'Trek!

Follow the CodeTrek to control the LED with PWM.

Run It!

Try changing the `LED_ON` duty cycle to see how the LED dims.

- duty_cycle=`0` → 0% ON time each period
- duty_cycle=`2**15-1` → 50% ON time each period
- duty_cycle=`2**16-1` → 100% ON time each period

CodeTrek:

```
1  from codex import *
2  from time import sleep_ms
3
4  # Constants for peripherals
```

```
5  LED_ON = 2**10 #duty_cycle = 2^10
```

Change the value of LED_ON to 2^10, or 1024.

- This is a low-power duty_cycle for the LED.

```
6  LED_OFF = 0
```

Change the value of LED_OFF to 0.

- This is a **no-power** duty_cycle for the LED.

```
7
8  # Set up the peripherals
9  led = exp.pwm_out(exp.PORT0)
```

Change the LED setup to use **PWM** as an output peripheral.

- The LED now accepts integer values instead of True and False.

```
10
11 def set_led(val):
12     led.duty_cycle = val
```

Change the way the LED turns on.

- Use led.duty_cycle instead of the value.

```
13
14 # Main program
15 while True:
16     set_led(LED_ON)
```

Change the argument in the function call to an integer value for duty_cycle.

- You changed LED_ON to an integer, so you can use it as the argument.

Hints:

- ## Real-world applications

  🔧PWM is used in many applications such as:

  - Computer Fans
  - Lights
  - DC Motor Controls
  - Electric Cookers
  - Voltage Regulators
  - Audio Effects

- ## That's a lot of bits!

  The CodeX has a 16-bit capacity.

  - That is 2^{16} or 65,536 bits.
  - It uses values from 0-6553 5.

Goals:

- Set up the **LED** by assigning the variable `led` as the output of `exp.pwm_out(exp.PORT0)`.
- In the `set_led()` function, assign `led.duty_cycle` as `val`.
- Call `set_led()`.

Tools Found: Loops, PWM, Variables, Functions

Solution:

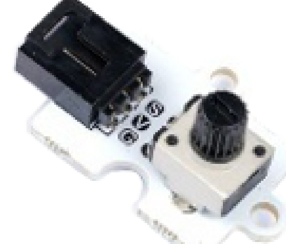
```
1  from codex import *
2  from time import sleep_ms
3
4  # Constants for peripherals
5  LED_ON = 2**10  # duty_cycle = 2^10
6  LED_OFF = 0
7
8  # Set up the peripherals
9  led = exp.pwm_out(exp.PORT0)
10
11 def set_led(val):
12     led.duty_cycle = val
13
14 # Main program
15 while True:
16     set_led(LED_ON)
```

Objective 4 - Analog Input - The Potentiometer

Your code can *dim* the LED, but..

It would be nice to be able to dim the LED with an external peripheral!

How about an **analog** input?



Concept: *Analog Input Peripherals*

A peripheral used for **analog input** returns a value from 0 to $2^{16} - 1$.

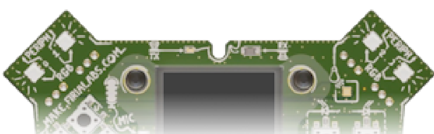
- Many analog inputs can put out 5 volts.
- The CodeX ADC can only show up to 2.8 volts.
- So, the voltage from the analog sensor needs to be cut in half.
- Use a **resistor voltage divider** to cut the voltage from the analog sensor to the CodeX ADC.

The simplest analog sensor in the LiftOff kit is the potentiometer.

- The potentiometer returns a value from 0 to $2^{16} - 1$.
- Rotating the **potentiometer** changes the **analog** value.
- The potentiometer needs the resistor voltage divider to limit the ADC input.

Connect Peripheral

Connect the **divider** to **PORT1** and the **potentiometer** to the **divider** on your CodeX.



Set up the potentiometer as an analog peripheral.

```
potentiometer = exp.analog_in(exp.PORT1)
```

Ooh! This is perfect!!

- The potentiometer returns a value between 0 and $2^{16} - 1$.
- The LEDs duty_cycle takes a value between 0 and $2^{16} - 1$.
- *Soon you'll be rotating the knob to dim the LED!*

Check the 'Trek!

Follow the CodeTrek to set up the potentiometer and use it to control the LED.

Physical Interaction

Rotate the knob and run your program.

Try *both ends* of the potentiometer's range by turning the knob *clockwise* and *counterclockwise.*

- You should be able to see a difference!
- But maybe not as much as you'd expect...
    - Your eye's *perception* of brightness doesn't exactly match the power level of the light (lumens).

CodeTrek:

```
1  from codex import *
2  from time import sleep_ms
3
4  # Constants for peripherals
5  LED_ON = 2**10  #duty_cycle = 2^10
6  LED_OFF = 0
7
8  # Set up the peripherals
9  led = exp.make_pwm(exp.PORT0)
10 potentiometer = exp.analog_in(exp.PORT1)
```

Set up the potentiometer as an analog input peripheral.

```
11
12 def set_led(val):
13     led.duty_cycle = val
14
15 # Main program
16 while True:
17     set_led(potentiometer.value)
```

Use the return value of the potentiometer as the argument for `set_led()`.

- The potentiometer value will be the LED's duty_cycle.

Hints:

- ## Analog to Digital

  An analog device measures continuously, so the data needs to be converted to a digital number for the computer.

  Want to see what is happening with the potentiometer? Add a `print()` statement to your code and watch the console to see the potentiometer's values.

```
def set_led(val):
    print("Reading", val)
    led.duty_cycle = val
```

- ## Analog limits

  The potentiometer may not get all the way down to 0 or up to 65535 ($2^{16} - 1$).

Goals:

- Set up the **potentiometer** by assigning the variable `potentiometer` as the output of `exp.analog_in(exp.PORT1)`.
- Call `set_led()` and supply the argument `potentiometer.value`.

Tools Found: CPU and Peripherals, Analog to Digital Conversion, LiftOff Kit, Variables, Keyword and Positional Arguments

Solution:

```
1  from codex import *
2  from time import sleep_ms
3
4  # Constants for peripherals
5  LED_ON = 2**10  # duty_cycle = 2^10
6  LED_OFF = 0
7
8  # Set up the peripherals
9  led = exp.pwm_out(exp.PORT0)
10 potentiometer = exp.analog_in(exp.PORT1)
11
12 def set_led(val):
13     led.duty_cycle = val
14
15 # Main program
16 while True:
17     set_led(potentiometer.value)
```

***Objective 5* - Motion Detector!!**

Energy Conservation.

You can save even more energy if you only turn on the lights when people are in the room.

But how can you detect when people are in the room?

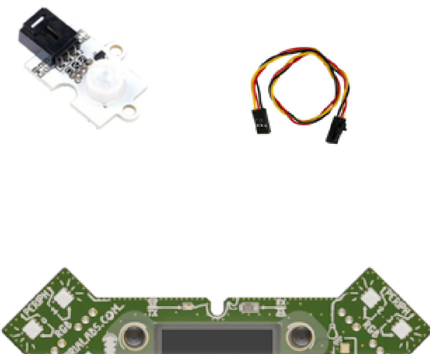
- You can start by sensing **motion**.

The PIR sensor in the LiftOff Kit is a Passive Infrared (PIR) motion sensor:

- It detects changes in **infrared (IR) light** from objects in its **field of view**.
- It can tell you that something ***moved!***
- ...but not *what* moved. Just assume it's an astronaut :-)

Connect Peripheral

Connect the **motion sensor** to **PORT2** on your CodeX.



Most sensors in the LiftOff kit provide 🔧analog values. But the **motion sensor** is 🔧digital.

Like the button and switch, the **motion sensor** is read with `exp.digital_in(exp.PORT2)`.

The **motion sensor** will output:

- `True` when it detects motion
- `False` when there is no motion

NOTE: The sensor will remain `True` for a brief period after motion has stopped.

Check the 'Trek!

Follow the CodeTrek to set up the motion sensor and use it to control the LED.

Physical Interaction: *Try it!*

- Wave your arm to turn *on* the LED.
- Be still... wait until your LED turns *off*.
- Change the potentiometer by turning the knob while the LED is on.

NOTE: You may need to block the sensor if there is a lot of motion in your area.

CodeTrek:

```
1  from codex import *
2  from time import sleep_ms
3
4  # Constants for peripherals
5  LED_ON = 2**10            # duty_cycle = 2^10
6  LED_OFF = 0
7  MOTION_DETECTED = True    # Motion Sensor
```

Define a constant for the motion sensor that will mean *motion detected*.

- You used constants like this for the switch and button in **LiftOff**.

```
8
9  # Set up the peripherals
10 led = exp.pwm_out(exp.PORT0)
11 potentiometer = exp.analog_in(exp.PORT1)
12 motion_sensor = exp.digital_in(exp.PORT2)
```

Set up the motion sensor as a digital input peripheral.

```
13
14 def set_led(val):
```

```
15      led.duty_cycle = val
16
17  while True:
18      if motion_sensor.value == MOTION_DETECTED:
19          set_led(potentiometer.value)
20      else:
21          set_led(LED_OFF)
```

Add an `if` statement to the `while True` loop

- Use the return value of the motion sensor in the condition.
- Turn on the LED when motion is detected.
- Use the potentiometer's value for the duty_cycle.
- To turn off the LED, the duty_cycle is `0`.

Hints:

- ## Blue Light

  The motion sensor has its own **Status LED**!

  - Do you see a *blue LED* light up next to the connector?
  - It is showing you the status of ***motion detection.***
  - *That could be a handy debugging tool!*

- ## Turn off the light!

  Remember that

  - duty_cycle=`0` → 0% ON time each period

  You changed the value of `LED_OFF` to `0`.

  - So to turn **OFF** the LED, use `LED_OFF` as the argument:

    ```
    set_led(LED_OFF)
    ```

Goals:

- Set up the **motion_sensor** by assigning the 🔧variable `motion_sensor` as the output of `exp.digital_in(exp.PORT2)`.
- Define the 🔧constant `MOTION_DETECTED`.
- Use an `if` statement with the 🔧condition `motion_sensor.value == MOTION_DETECTED`.
- Call `set_led(potentiometer.value)` and `set_led(LED_OFF)`.

Tools Found: LiftOff Kit, Analog to Digital Conversion, Binary Numbers, Variables, Constants, bool

Solution:

```
1   from codex import *
2   from time import sleep_ms
3
4   # Constants for peripherals
5   LED_ON = 2**10            # duty_cycle = 2^10
6   LED_OFF = 0
7   MOTION_DETECTED = True    # Motion Sensor
8
9   # Set up the peripherals
10  led = exp.pwm_out(exp.PORT0)
11  potentiometer = exp.analog_in(exp.PORT1)
12  motion_sensor = exp.digital_in(exp.PORT2)
13
14  def set_led(val):
15      led.duty_cycle = val
16
```

```
17 # Main program
18 while True:
19     if motion_sensor.value == MOTION_DETECTED:
20         set_led(potentiometer.value)
21     else:
22         set_led(LED_OFF)
```

Quiz 2 - Check for Understanding: Analog vs Digital

Question 1: Which peripheral is **analog**?

- ✓ Potentiometer
- ✗ Motion Sensor
- ✗ White LED
- ✗ Button

Question 2: What are the possible values for the **Motion Sensor**?

- ✓ `True` and `False`
- ✗ 0 and 1
- ✗ 0 to 65535
- ✗ All positive integers

Objective 6 - Delayed Shutdown

Give it some time!

It would probably be annoying if your lights shut off every time you stopped moving.

What if you want the lights to stay on a while each time motion is detected?

- You could use a delay like `sleep_ms()` to keep the LED on.
- Start by defining how long you want the lights to stay on in **milliseconds**.

This constant will keep the lights ON for 10 seconds (10,000 milliseconds):

```
ON_TIME = 10*1000 # 10 * 1000 milliseconds or 10 seconds
```

Check the 'Trek!

Follow the CodeTrek to define a constant for how long to keep the LED on and use it in a delay.

Physical Interaction: *Move it!*

CodeTrek:

```
1  from codex import *
2  from time import sleep_ms
3
4  # Constants for peripherals
5  LED_ON = 2**10           # duty_cycle = 2^10
6  LED_OFF = 0
7  MOTION_DETECTED = True   # Motion Sensor
8  ON_TIME = 10 * 1000      # 10 * 1000 milliseconds (10 seconds)
```

Define a constant for how long to keep the light on after motion is detected.

- 10 seconds is a suggestion. For testing, you may want a shorter delay.
- NOTE: 1000 milliseconds in 1 second

```
9
10 # Set up the peripherals
11 led = exp.make_pwm(exp.PORT0)
12 potentiometer = exp.analog_in(exp.PORT1)
13 motion_sensor = exp.digital_in(exp.PORT2)
14
15 def set_led(val):
16     led.duty_cycle = val
17
18 while True:
19     if motion_sensor.value == MOTION_DETECTED:
20         set_led(potentiometer.value)
21         sleep_ms(ON_TIME)
```

Use the constant `ON_TIME` in a delay to keep the LED on.

```
22     else:
23         set_led(LED_OFF)
```

Goals:

- Define the constant `ON_TIME`.
- Call `sleep_ms(ON_TIME)` *after* `set_led(potentiometer.value)`.

Tools Found: Timing, Constants

Solution:

```
1  from codex import *
2  from time import sleep_ms
3
4  # Constants for peripherals
5  LED_ON = 2**10          # duty_cycle = 2^10
6  LED_OFF = 0
7  MOTION_DETECTED = True  # Motion Sensor
8  ON_TIME = 10 * 1000     # 10 * 1000 milliseconds (10 seconds)
9
10 # Set up the peripherals
11 led = exp.pwm_out(exp.PORT0)
12 potentiometer = exp.analog_in(exp.PORT1)
13 motion_sensor = exp.digital_in(exp.PORT2)
14
15 def set_led(val):
16     led.duty_cycle = val
17
18 # Main program
19 while True:
20     if motion_sensor.value == MOTION_DETECTED:
21         set_led(potentiometer.value)
22         sleep_ms(ON_TIME)
23     else:
24         set_led(LED_OFF)
```

Objective 7 - Time to Blackout

Make your *LED dimming knob* work all the time!

The `sleep_ms()` in your loop is blocking your ability to change the brightness.

- The `sleep_ms()` function doesn't return until *after* the requested time has elapsed.

- So your code is ***stopped in its tracks*** while it waits for `sleep_ms()` to finish!

If you want the **knob** to work all the time, your code needs to continuously **read** it in a 🔧loop.

- Is there a way to *read* the ***time*** while you're looping?

Concept: *Non-Blocking Timer*

Going to sleep **blocks** your code from doing anything else!

- Luckily the **Time** library module provides a function that keeps track of time in embedded code.
- Your code can **check the time** whenever it needs to see how much time has elapsed.
  - Like glancing at your watch... *"Is it time yet?"*

From the moment you connect the CodeX, the **Timer** is *always running*.

Here's a built-in **function** that returns exactly how many *milliseconds* have elapsed since the last 🔧reboot:

```
# Return number of milliseconds since reboot
time.ticks_ms()
```

Keeping track of time

*Imagine **you** are the computer...*

When motion is detected, you need to glance at your watch.

- *"Okay, 10 seconds from now I should turn the LED off."*
  - Um, better write down the *"turn-off time"* somewhere.
- Then keep checking the **knob** like always...
- But also keep an eye on your watch.
  - And `if` the time is up, ***turn the LED off!***

So there's your **Algorithm**. Now you just have to *code* it!

Check the 'Trek!

Follow the CodeTrek to use a loop to check the time ***and*** adjust the brightness.

Physical Interaction: *Try it!*

Run the code. Try changing the brightness of your LED during the ON_TIME window:

- After motion is detected but before the LED turns OFF.

CodeTrek:

```
1  from codex import *
2  import time
```

Import the entire **Time** library module.

```
3
4  # Constants for peripherals
5  LED_ON = 2**10  #duty_cycle = 2^10
6  LED_OFF = 0
7  MOTION_DETECTED = True
8  ON_TIME = 10 * 1000  # 10 * 1000 milliseconds
9
10 # Set up the peripherals
11 led = exp.pwm_out(exp.PORT0)
12 potentiometer = exp.analog_in(exp.PORT1)
```

```
13  motion_sensor = exp.digital_in(exp.PORT2)
14
15  def set_led(val):
16      led.duty_cycle = val
17
18  # Main program
19  turn_off_time = time.ticks_ms()
```

Define a variable for ending time, and give it an initial value of the current time.

```
20
21  while True:
22      if motion_sensor.value == MOTION_DETECTED:
23          # Set the turn_off_time from when motion is detected
24          turn_off_time = time.ticks_ms() + ON_TIME
```

If motion is detected, update `turn_off_time` to the current time plus `ON_TIME`

- This is the amount of time to delay, or keep the light on.

```
25          # Use a while loop to read the knob during ON_TIME
26          while time.ticks_ms() < turn_off_time:
27              set_led(potentiometer.value)
```

Use another loop to read the knob's value while keeping the light on.

- The loop replaces the `sleep_ms()` statement.
- This is a "loop within a loop" or a nested loop.
- Watch your indenting!

```
28      else:
29          set_led(LED_OFF)
30
```

Hints:

- ## Tracking time

  The `ticks()` and `ticks_ms()` functions are very useful for marking the passage of time in embedded code.

  - These counters start at **zero** when your device boots, and keep counting **up** from there while it is running.
  - To learn more about these functions, and others in the time module, check out the toolbox.

- ## How much time?

  The crew will probably want the lights to stay on longer than 10 seconds.

  For testing your code, this is a good length of time. Then, after you know your code works, you can easily change the value of the constant to the actual amount of time to leave the lights on.

Goals:

- Import the *entire* `time` library.
- Define the variable `turn_off_time` with the *initial* value `time.ticks_ms()`.
- Use a `while` loop with the condition `time.ticks_ms() < turn_off_time`.

Tools Found: Loops, Reboot, import, Variables, bool

Solution:

```
1  from codex import *
2  import time
```

```
3
4  # Constants for peripherals
5  LED_ON = 2**10            # duty_cycle = 2^10
6  LED_OFF = 0
7  MOTION_DETECTED = True    # Motion Sensor
8  ON_TIME = 10 * 1000       # 10 * 1000 milliseconds (10 seconds)
9
10 # Set up the peripherals
11 led = exp.pwm_out(exp.PORT0)
12 potentiometer = exp.analog_in(exp.PORT1)
13 motion_sensor = exp.digital_in(exp.PORT2)
14
15 def set_led(val):
16     led.duty_cycle = val
17
18 # Main program
19 turn_off_time = time.ticks_ms()
20 while True:
21     if motion_sensor.value == MOTION_DETECTED:
22         # Set the new turn_off_time from when motion is detected
23         turn_off_time = time.ticks_ms() + ON_TIME
24         # Use a while loop to read the knob during ON_TIME
25         while time.ticks_ms() < turn_off_time:
26             set_led(potentiometer.value)
27     else:
28         set_led(LED_OFF)
29
```

Mission 3 Complete

You've completed project *Conserve Energy!*

...and you helped save enough energy to get the crew to Mars.

What else does the mission require?

- ***Plenty!*** And you're developing the *coding skills* to make it happen!

Mission 4 - Hatch Lock!

This project introduces the NeoPixel ring and will show you how to light up different colors!

Mission Briefing:

The crew's shuttle will dock with a larger supporting craft that launched ahead of the mission.

The supporting craft will have extra fuel, supplies, and larger quarters for the crew to live in.

There are eight locks that engage on the shuttle's hatch to get a proper seal with the supporting craft.

- The locks engage when a large mechanical lever is moved.
- The crew will need an indication that all eight locks have engaged.
- The mechanical engineering team has discovered an issue where individual locks only engage 85% of the time.

Project: Hatch Lock will guide you through triggering the locks and displaying individual lock information.

Project Goals:

- Learn about using RGB colors with NeoPixels.
- Understand uses for microswitches.
- Inform the crew when the hatch is properly secured to the supporting craft!!

Objective 1 - NeoPixels

The crew needs to know the status of the 8 locks on the hatch!

You can use the NeoPixel ring to display the status of each lock in ***color!***

- The NeoPixel ring has 8 separate RGB (Red, Green, Blue) LEDs called **pixels**.
- Each pixel can be set to a unique color.
- The pixels can be individually dimmed.

Check the 💡 Hints for fun facts about NeoPixels

Connect Peripheral

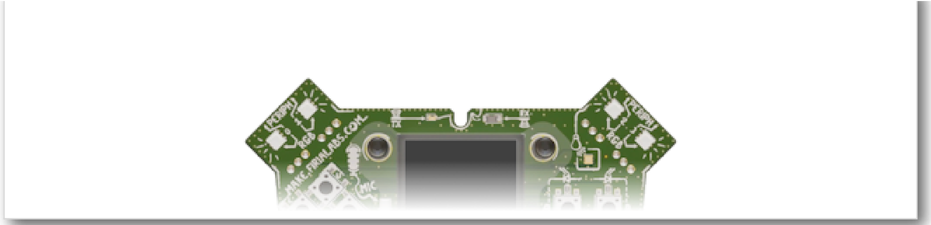
Connect the **8 RGB NeoPixel ring** to **PORT0** on your CodeX. Use the three loose wires provided in the peripheral kit.

The back side of the NeoPixel ring has three labels:

- **G** or *ground* is connected to the **BLACK** wire.
- **V** or *voltage* is connected to the **ORANGE** wire.
- **DI** or *data input* is connected to the **YELLOW** wire.

  On the CodeX, the YELLOW wire goes to **S**, the ORANGE in the middle, and the BLACK to the **G**.

Disconnect all other peripherals from the CodeX.



Setup your NeoPixel on PORT0

Two pieces of information are needed when you set up the NeoPixel ring:

- The port the NeoPixel ring is connected to.
- The number of **pixels** connected.

Use this code:

```
np = neopixel.NeoPixel(exp.PORT0, 8)  # 8 pixels on PORT0
```

It's time to turn on the first LED on the NeoPixel ring.

You can set the *color* of a *pixel*

- The NeoPixel object `np` works similar to a Python list.
  - The list will hold an individual color for each of the 8 pixels.
  - Remember that you index a list item using brackets[].
  - Also recall that the index count starts at ***zero!***

Create a New File!

Use the **File** → **New File** menu to create a new file called ***HatchLock***.

Check the 'Trek!

Follow the CodeTrek to set the first pixel color and turn it on.

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Set up the peripherals
5  power.enable_periph_vcc(True)
```

The NeoPixel ring requires a lot of power, so you will manually turn the power on.

- Remember to import the libraries you will need.

```
6  np = neopixel.NeoPixel(exp.PORT0, 8)
```

Set up the NeoPixel ring on PORT0

- It has 8 pixels on the ring.

```
7
8  while True:
9      # Set the color for pixel 0
10     np[0] = (20, 0, 0)
```

Set the color for the first pixel.

- The first index in the `np` pixel list is `0`.
- The RGB color is (20, 0, 0), which is RED.

```
11
```

Hints:

- ## Connecting the NeoPixel ring

  Most of your connectors have wires the same three colors:
- Black for **GROUND**
- Red for **POWER**
- Yellow for **SIGNAL**

The connector for the NeoPixel ring may have different colors. That is okay. Just make sure to connect the same color to the same pin on the peripheral and the CodeX.

Optionally, you can connect the NeoPixel ring directly to the CodeX PORT.

- ## Fun Facts about NeoPixels

  NeoPixels can come in all sorts of shapes and sizes!
- There are NeoPixel strips with 1000s of different pixels

- They can be laid out in strips, rectangles, circles, etc.

- ## More Fun Facts

  NeoPixels are most commonly used for:
- Room lighting
- Hobbyist electronic displays
- Clothing
- Jewelry

- Accessories *(like a light-up snowboard!)*

- ## 0 or 1

  In the code, pixel `0` is the one nearest the *connector* of your *Neopixel Ring*.
- If you look closely you'll see the pixels are numbered *clockwise* around the ring.
- But the numbering starts with **1**!
  - Don't let that confuse you. In software, the *first* item is usually index `0`.

Humans usually count "1, 2, 3, ..."

Computers count "0, 1, 2, ..."

Goals:

- Create a new file named `HatchLock`.

- Set up the **NeoPixel** by assigning the 🔧variable `np` as the output of `neopixel.NeoPixel(exp.PORT0, 8)`.

- Assign the color of the first pixel as `(20, 0, 0)`.

Tools Found: list, Variables

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Set up the peripherals
5  power.enable_periph_vcc(True)
6  np = neopixel.NeoPixel(exp.PORT0, 8)
7
```

```
 8  while True:
 9      # Set the color for pixel 0
10      np[0] = (20, 0, 0)
11
```

Objective 2 - How do NeoPixels Work?

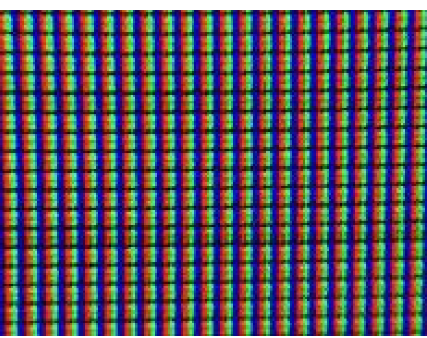
Each pixel is made up of three LEDs.

- **One is Red**
- **One is Green**
- **One is Blue**

Concept

NeoPixels create colors similar to how display screens do it in phones and TVs.

- If you could look close enough at those displays you'd see something like this.
- *Lots* of tiny RED/GREEN/BLUE *dots!*
  - A 4K display has 3840x2160 = 8,294,400 pixels!



Color Me a tuple of ints!

LED displays show color by blending (Red, Green, Blue) together.

- Python uses a data type called a tuple to define a **color**
- A tuple is a Python data type that holds a sequence of values.
- A **tuple** is similar to a list; it can hold multipe values.
  - A tuple is perfect for defining `color = (red, green, blue)`.
  - A tuple is defined with parenthesis () instead of square brackets [].
  - Unlike in a list, the values in a tuple can NOT be changed.

Can you *see* the *R-G-B* ?

When a Neopixel is *brightly lit*, the colors blend together visually.

- But if the LEDs are *very dim*, maybe you can see them individually!

Change the value of the first pixel

```
np[0] = (2, 2, 2)
```

Run It!

Look closely at the first pixel. Can you see the 3 LEDs inside?

When you are setting an RGB value you are actually setting the brightness of the 3 LEDs.

- The range for a single LED is `0` to `255`
  - `0` is **OFF**
  - `255` is *full brightness* **ON**

Caution: *Make sure you don't look directly at bright LEDs!*

The rest of the code in this project will keep the brightness to **20 or less** for an individual LED. A brightness of 255 is **VERY** bright and you should not look directly at it!

You can use variables to set the brightness of each LED. Then combine the variables into a tuple.

Check the 'Trek!

Follow the CodeTrek to define a tuple for the color.

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Set up the peripherals
5  power.enable_periph_vcc(True)
6  np = neopixel.NeoPixel(exp.PORT0, 8)
7
8  # Brightness variables
9  red = 0
10 green = 20
11 blue = 20
```

Use three color variables to set the LED's color.

- The colors are `red`, `green` and `blue`.
- NOTE: Variable names should not be capitalized! Use all lowercase letters.

```
12 rgb_color = (red, green, blue)
```

Use the three color variables to define a tuple for `rgb_color`.

- What color do you think this will display?

```
13
14 while True:
15     # Set the color for pixel 0
16     np[0] = rgb_color
```

Assign the tuple to the first pixel.

```
17
```

Hints:

- Each pixel has 3 LEDs: Red, Green and Blue.
  - You control the brightness of each LED with a value from `0` (OFF) to `255` (full brightness ON).
  - That is a total of 256 levels of brightness. This requires 8 bits.
  - 2^8 = 256 (or values from 0 to 255)
- Experiment with different values to see the difference in the brightness level.
  - Pick one color, like Blue.

- Start with a small brightness level, like 5.
- Run the code.
- Change the brightness level a little at a time and observe the brightness change.

```
np[0] = (0, 0, 5)
```

CAUTION: Don't use a brightness level above 2 0.

Goals:

- Define `red`, `green`, and `blue` 🔧variables.
- Define `rgb_color` as a tuple of `red`, `green` and `blue`.
- Assign `rgb_color` to the first pixel.

Tools Found: tuple, int, Data Types, list, Variables

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Set up the peripherals
5  power.enable_periph_vcc(True)
6  np = neopixel.NeoPixel(exp.PORT0, 8)
7
8  # Brightness variables
9  red = 0
10 green = 20
11 blue = 20
12 rgb_color = (red, green, blue)
13
14 while True:
15     # Set the color for pixel 0
16     np[0] = rgb_color
17
```

Quiz 1 - Check Your Understanding: NeoPixels

Question 1: How many brightness levels are there for each NeoPixel?

✓ 256

✗ 2

✗ 255

✗ 20

Question 2: How many 🔧binary **bits** are used for each NeoPixel color?

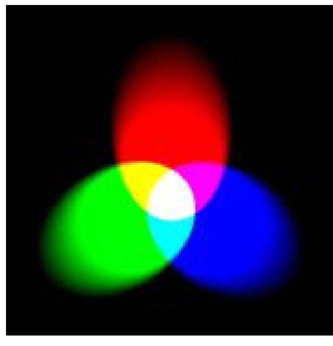
✓ 8

✗ 2

✗ 20

✗ 16

Objective 3 - Color Blend and Change!



You already know that there are 3 LEDs in a single NeoPixel, so it is easy to make a pixel either *red*, *green*, or *blue*.

What if you want yellow?

- The LEDs are close enough together that the colors can blend together and appear as a single color.

Here are some common RGB colors:

```
RGB_RED = (20, 0, 0)
RGB_GREEN = (0, 20, 0)
RGB_BLUE = (0, 0, 20)
RGB_MAGENTA = (20, 0, 20)
RGB_YELLOW = (20, 20, 0)
RGB_CYAN = (0, 20, 20)
RGB_WHITE = (20, 20, 20)
RGB_OFF = (0, 0, 0)
```

On the NeoPixel ring, the eight pixels are numbered 0 to 7.

- You have already accessed the first pixel using `np[0]`.
- You can access the other 7 pixels in the same way, with their **index** number.

Check the 'Trek!

Follow the CodeTrek to turn on all 8 pixels.

- Define constants for two colors.
- Set half the pixels to one color.
- Delay for 1 second.
- Set the other half to the second color.

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Set up the peripherals
5  power.enable_periph_vcc(True)
6  np = neopixel.NeoPixel(exp.PORT0, 8)
7
8  # brightness variables and constants
9  red = 10
10 green = 5
11 blue = 20
12 rgb_color = (red, green, blue)
13 RGB_RED = (20, 0, 0)
14 RGB_YELLOW = (20, 20, 0)
```

Define constants for at least two colors.

- You can use any two colors.

```
15
16 while True:
17     # Set the color for the even-numbered pixels
18     np[0] = RGB_RED
19     np[2] = RGB_RED
20     np[4] = RGB_RED
21     np[6] = RGB_RED
```

Set the even-numbered pixels to the first color.

```
22      sleep(1)
```

Delay for 1 second.

```
23      # Set the color for the odd-numbered pixels
24      np[1] = RGB_YELLOW
25      np[3] = RGB_YELLOW
26      np[5] = RGB_YELLOW
27      np[7] = RGB_YELLOW
28      sleep(1)
```

Set the odd-numbered pixels to the second color.

- Delay for 1 second.

Hints:

- ## Make Your Own Colors

  Try different combinations of brightness levels for red, green and blue. When you find a color you like you can define a constant for it. Then you can easily use the constant as the color value for any (or all) of the pixels.

  For example:

  ```
  RGB_LAVENDER = (10, 5, 20)
  np[7] = RGB_LAVENDER
  ```

- # CONSTANT Colors

  Did you notice the names of the color constants are NOT **RED**, **GREEN** or **BLUE**?

  - **RED**, **GREEN** and **BLUE**, as well as many other colors, are already defined in the **codex** library that you imported.
  - Use different descriptive names for your colors, like **RGB_RED** and **RGB_BLUE**.

Goal:

- Assign a color to each of the **eight** NeoPixels.

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Set up the peripherals
5  power.enable_periph_vcc(True)
6  np = neopixel.NeoPixel(exp.PORT0, 8)
7
8  # brightness variables and constants
9  red = 10
10 green = 5
11 blue = 20
12 rgb_color = (red, green, blue)
13 RGB_RED = (20, 0, 0)
14 RGB_YELLOW = (20, 20, 0)
15
16 while True:
17     # This is one solution for Obj. 3
18     # Students can use more than 2 colors, and a
19     # delay isn't required for validation.
20     # Set the color for the even-numbered pixels
21     np[0] = RGB_RED
22     np[2] = RGB_RED
23     np[4] = RGB_RED
24     np[6] = RGB_RED
25     sleep(1)
26     # Set the color for the odd-numbered pixels
27     np[1] = RGB_YELLOW
```

```
28      np[3] = RGB_YELLOW
29      np[5] = RGB_YELLOW
30      np[7] = RGB_YELLOW
31      sleep(1)
```

Objective 4 **- Fill all the positions!**

A important component of coding is organizing the code into 🔧functions.

Your code inside the `while True:` loop is getting long!

- Let's set all of the pixels to a single color with a 🔧function!
- Add a 🔧function `set_all_pixels()` to your code.
- It should use one 🔧parameter named `rgb_color`.

```
def set_all_pixels(rgb_color):
    np[0] = rgb_color
    np[1] = rgb_color
    np[2] = rgb_color
    np[3] = rgb_color
    np[4] = rgb_color
    np[5] = rgb_color
    np[6] = rgb_color
    np[7] = rgb_color
```

Then update the `while True:` loop to call the function.

- ***NOTE:*** All pixels will now be the same color.

```
while True:
    set_all_pixels(rgb_color)
    sleep(1)
```

The function has a lot of *repetitive* code to just set the pixels!

What can you use to reduce the size of your code?

You can use a `for` 🔧loop!!!

- The 🔧loop will need to iterate 8 times.
  - Using `range(8)` will start the loop ***variable*** at `0` and end with `7`.
  - Each iteration of the loop will set a pixel.
  - Use the loop variable ***pixel*** as the ***index*** for the `np` list.

Check the 'Trek!

Follow the CodeTrek to use a `for` loop in the `set_all_pixels()` block.

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Set up the peripherals
5  power.enable_periph_vcc(True)
6  np = neopixel.NeoPixel(exp.PORT0, 8)
7
8  # brightness variables and constants
9  red = 10
10 green = 5
11 blue = 20
12 rgb_color = (red, green, blue)
13 RGB_RED = (20, 0, 0)
14 RGB_YELLOW = (20, 20, 0)
15
```

```
16  # Set all pixels to one color
17  def set_all_pixels(rgb_color):
```

Define the function `set_all_pixels()` with one parameter `rgb_color`.

```
18      for pixel in range(8):
19          np[pixel] = rgb_color
```

Use a `for` loop to set all the pixels one color.

- `pixel` is the **loop variable** and the index for `np`.
- The loop variable starts at `0` and ends at `7`, which corresponds to the numbers of the pixels.

```
20
21  while True:
22      # Set the colors for all pixels
23      set_all_pixels(RGB_RED)
24      sleep(1)
```

Call the function `set_all_pixels()` in the `while True:` loop.

- Use a constant or `rgb_color` for the argument.

```
25      # Optional
26      set_all_pixels(RGB_YELLOW)
27      sleep(1)
```

This step is optional.

- You can call `set_all_pixels()` a second time with a different argument.

Goals:

- Define the function `set_all_pixels(rgb_color)`.
- Use a `for` loop that iterates over the sequence `range(8)`.
- Call `set_all_pixels()`.

Tools Found: Functions, Parameters, Arguments, and Returns, Loops, Iterable

Solution:

```
1   from codex import *
2   from time import sleep
3
4   # Set up the peripherals
5   power.enable_periph_vcc(True)
6   np = neopixel.NeoPixel(exp.PORT0, 8)
7
8   # brightness variables
9   red = 10
10  green = 5
11  blue = 20
12  rgb_color = (red, green, blue)
13  RGB_RED = (20, 0, 0)
14  RGB_YELLOW = (20, 20, 0)
15
16  # Set all pixels to one color
17  def set_all_pixels(rgb_color):
18      for pixel in range(8):
19          np[pixel] = rgb_color
20
21  while True:
```

```
22      # Set the color for all pixels
23      set_all_pixels(RGB_RED)
24      sleep(1)
25      # Optional
26      set_all_pixels(RGB_YELLOW)
27      sleep(1)
```

Objective 5 **- Random Fun!**

Unlock the potential of the NeoPixels - make a disco ball!

Even space travelers might need a little fun!!

You have defined colors using RGB and set the pixels to those colors. But what about ***random*** colors?

You can use a loop to generate a different **random** color for each **pixel**, creating a cool disco ball.

You already have variables for `red`, `green`, and `blue`, as well as `rgb_color`. Use these variables in your loop to get a random color for each pixel.

Check the 'Trek!

Follow the CodeTrek to define a function `disco_ball()` that generates a random color for each pixel.

Now *cue the music* and get your *code* on!

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  from random import randint
```

Import `randint` from the `random` module.

- To generate a random color, you need integers.

```
4
5  # Set up the peripherals
6  power.enable_periph_vcc(True)
7  np = neopixel.NeoPixel(exp.PORT0, 8)
8
9  # brightness variables and constants
10 RGB_RED = (20, 0, 0)
11 RGB_YELLOW = (20, 20, 0)
12
13 # Set all pixels a random color
14 def disco_ball():
```

Create a new function for the `disco_ball()`.

```
15     for pixel in range(8):
```

Use a `for` loop to generate random colors for each pixel.

- You can copy the `for` loop from `set_all_pixels()` and make changes.

```
16         red = randint(0, 20)
```

```
17          green = randint(0, 20)
18          blue = randint(0, 20)
19          rgb_color = (red, green, blue)
```

Move these lines of code from the ***brightness variables*** section above the function to inside the function.

- Be careful with your indenting!
- Change the assignment for each color to a random number.
- You will get a random brightness for red, green and blue.
- You don't want to go brighter than 20!

```
20          np[pixel] = rgb_color
```

Now that you have a random color, set the pixel to the color.

```
21
22  # Set all pixels to one color
23  def set_all_pixels(rgb_color):
24      for pixel in range(8):
25          np[pixel] = rgb_color
26
27  # Main program
28  while True:
29      # Call the disco_ball function
30      disco_ball()
31      sleep(0.2)
```

Call `disco_ball()` in the `while True:` loop.

- You can adjust the delay to get the disco affect you want.

Hints:

- ## Random Numbers

  The `random` library module in Python has several commands that will generate numbers in an unpredictable pattern.

  The `random` module must be imported before the commands can be accessed. If you are only using one command, you can import only that command `from random`.

- ## Random Number Commands

  Two commands that generate random integers are:

  - `randrange()`
    - Similar to the `range()` in a `for` loop.
    - It generates an integer between 0 and 1 less than the range.
    - **Example:** randrange(8) will return a number between 0 and 7.
  - `randint()`
    - Generates an integer that includes the low and high range.
    - **Example:** randint(1, 20) will return a number between 1 and 2 0.

Goals:

- Import `randint` from the `random` module.
- Define the function `disco_ball()`.
- Assign `red`, `blue`, and `green` to random values.

Tools Found: import, Functions

Solution:

```
1  from codex import *
2  from time import sleep
3  from random import randint
4
5  # Set up the peripherals
6  power.enable_periph_vcc(True)
7  np = neopixel.NeoPixel(exp.PORT0, 8)
8
9  # Brightness variables and constants.
10 RGB_RED = (20, 0, 0)
11 RGB_YELLOW = (20, 20, 0)
12
13 # Set all pixels a random color
14 def disco_ball():
15     for pixel in range(8):
16         red = randint(0, 20)
17         green = randint(0, 20)
18         blue = randint(0, 20)
19         rgb_color = (red, green, blue)
20         np[pixel] = rgb_color
21
22 # Set all pixels to one color
23 def set_all_pixels(rgb_color):
24     for pixel in range(8):
25         np[pixel] = rgb_color
26
27 # Main program
28 while True:
29     # Call the disco_ball function
30     disco_ball()
31     sleep(0.2)
```

Quiz 2 **- Check Your Understanding: Random LEDs**

Question 1: What color LED is lit up in the RGB value (0, 20, 0)?

✓ Green

✗ Red

✗ Blue

✗ Yellow

Question 2: How many LEDs make up a single pixel in a NeoPixel?

✓ 3

✗ 8

✗ 1

✗ 10

Question 3: What is the range of numbers possible with `randint(1, 10)`?

✓ All integers between, and including, 1 through 10

✗ All integers between, and including, 1 through 9

✗ The integers 1 and 10 only

✗ All real numbers between 1 and 10

Objective 6 **- Microswitch!**

That was fun... Now it is time to get back to work.

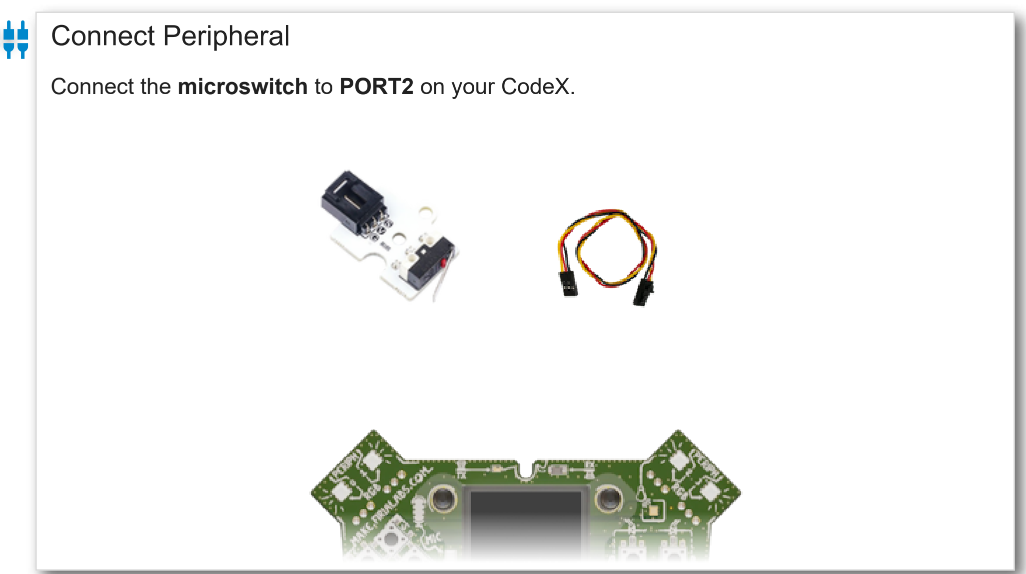
Your newfound NeoPixel skills are needed to show the status of 8 locks on the space shuttle's hatch!

You will also need one more tool: something to trigger the system to check the locks.

- You can use a ***microswitch!!***

Connect Peripheral

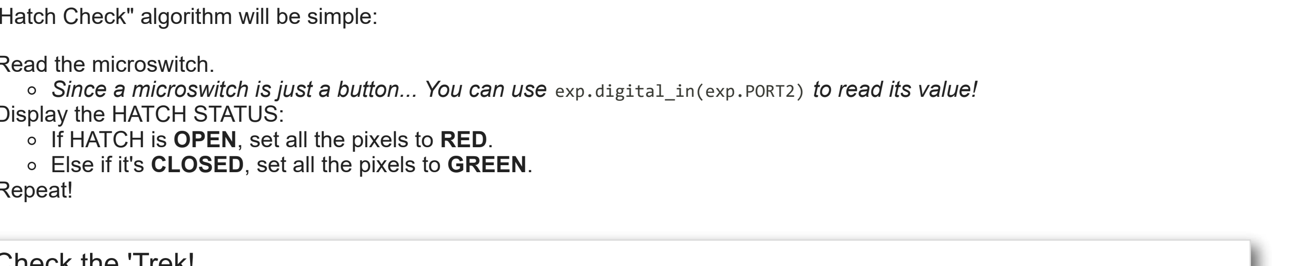
Connect the **microswitch** to **PORT2** on your CodeX.



Tell the crew if the hatch door is closed by lighting up the NeoPixels.

Your "Hatch Check" algorithm will be simple:

1. Read the microswitch.
   - *Since a microswitch is just a button... You can use* `exp.digital_in(exp.PORT2)` *to read its value!*
2. Display the HATCH STATUS:
   - If HATCH is **OPEN**, set all the pixels to **RED**.
   - Else if it's **CLOSED**, set all the pixels to **GREEN**.
3. Repeat!



Check the 'Trek!

Follow the CodeTrek to set up the **microswitch** and use it to turn all pixels **RED** if the hatch door is open and **GREEN** if the hatch is closed.

Physical Interaction: *Try it!*

Run the code. Press the **microswitch** to simulate closing the hatch!

- Microswitch open = **RED** pixels.
- Microswitch closed = **GREEN** pixels.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  from random import randint
4
5  # Set up the peripherals
6  power.enable_periph_vcc(True)
```

```
7  np = neopixel.NeoPixel(exp.PORT0, 8)
8  microswitch = exp.digital_in(exp.PORT2)
```

Set up the **microswitch** on **PORT2**.

- It is a digital input peripheral.

```
9
10 # Constants for the peripherals
11 HATCH_CLOSED = False    # Microswitch
12 #TODO: Define a constant for HATCH_OPEN
```

Define constants for HATCH_CLOSED and HATCH_OPEN.

- These constants will be used to compare with microswitch's return value.

```
13
14 # brightness variables and constants
15 RGB_RED = (20, 0, 0)
16 RGB_YELLOW = (20, 20, 0)
17 #TODO: Define a constant for RGB_GREEN
```

Define a constant for RGB_GREEN.

- If you don't already have this constant, define it now.
- If the hatch door is closed, the NeoPixels need to show **GREEN**.

```
18
19 # Set all pixels a random color
20 def disco_ball():
21     for pixel in range(8):
22         red = randint(0, 20)
23         green = randint(0, 20)
24         blue =randint(0, 20)
25         rgb_color = (red, green, blue)
26         np[pixel] = rgb_color
27
28 # Set all pixels to one color
29 def set_all_pixels(rgb_color):
30     for pixel in range(8):
31         np[pixel] = rgb_color
32
33 # Main program
34 while True:
35     # Read the microswitch and display the NeoPixels
36     if microswitch.value == HATCH_CLOSED:
37         set_all_pixels(RGB_GREEN)
38     else:
39         set_all_pixels(RGB_RED)
```

Use an `if` statement in the `while True:` loop to determine if the hatch door is closed.

- Compare the microswitch's return value to `HATCH_CLOSED`.
- If the hatch door is closed, set the pixels to **GREEN**.
- Else, set the pixels to **RED**.

Hints:

- ## Microswitch vs Button

  A **microswitch** is really just a *button* that's not intended for direct human interaction.

  - They are generally found in closed mechanical systems.

- ## Real-World Applications

  Microswitches are used in a vast number of products such as:

- Appliances
- Factory equipment
- Automobiles
- Touch detection for robots

- ## Control True/False

  The microswitch, like the button returns `True` when it is **up** (not pressed) and `False` when it is **down** (pressed).

  - Use a 🔧constant `HATCH_CLOSED` and assign it the value `False`, like you did for the switch.
  - Use another constant for `HATCH_OPEN` and assign it the value `True`.

Goals:

- Set up the **microswitch** by assigning the 🔧variable `microswitch` as the output of `exp.digital_in(exp.PORT2)`.
- Define 🔧constants `HATCH_OPEN` and `RGB_GREEN`.
- Use an `if` statement with the 🔧condition `microswitch.value == HATCH_CLOSED`.
- Call `set_all_pixels(RGB_GREEN)` *and* `set_all_pixels(RGB_RED)`.

Tools Found: Variables, Constants, bool

Solution:

```
1  from codex import *
2  from time import sleep
3  from random import randint
4
5  # Set up the peripherals
6  power.enable_periph_vcc(True)
7  np = neopixel.NeoPixel(exp.PORT0, 8)
8  microswitch = exp.digital_in(exp.PORT2)
9
10 # Constants for the peripherals
11 HATCH_CLOSED = False   # Microswitch
12 HATCH_OPEN = True      # Microswitch
13
14 # Brightness variables and constants.
15 RGB_RED = (20, 0, 0)
16 RGB_YELLOW = (20, 20, 0)
17 RGB_GREEN = (0, 20, 0)
18
19 # Set all pixels a random color
20 def disco_ball():
21     for pixel in range(8):
22         red = randint(0, 20)
23         green = randint(0, 20)
24         blue =randint(0, 20)
25         rgb_color = (red, green, blue)
26         np[pixel] = rgb_color
27
28 # Set all pixels to one color
29 def set_all_pixels(rgb_color):
30     for pixel in range(8):
31         np[pixel] = rgb_color
32
33 # Main program
34 while True:
35     # Read the microswitch and display the NeoPixels
36     if microswitch.value == HATCH_CLOSED:
37         set_all_pixels(RGB_GREEN)
38     else:
39         set_all_pixels(RGB_RED)
```

Objective 7 **- Problem with the Locks!**

The engineers have told you that the locks will only lock properly 85% of the time.

- That means there is a 15% failure rate.
- You need to show the crew which locks have failed!!
  - Right now your code **assumes** all 8 locks are working (GREEN) when the **microswitch** is pressed.
  - The *microswitch* just means **"Hatch Closed"**...
  - You need a new 🔧function to ***check the locks!***

*Modify your code to **accurately** simulate the **Hatch Locks!***

You will need two more 🔧functions.

- Define a function `def is_lock_failed()` that will return `True` if the lock fails to latch.
- Define a function `def check_all_locks()` that will check each lock and set each pixel individually.
  - Call this function when the **hatch** is **CLOSED**.
  - This function will call `is_lock_failed()` for each lock (0-7).

Check the 'Trek!

Follow the CodeTrek to define the two functions. Then modify the code to simulate the hatch locks.

Physical Interaction: *Try it!*

Run the code. Press and hold the microswitch to simulate closing the hatch!

Hmmm... something is wrong. The pixels should show up red or green, but you can see them flickering.

What is happening?

- When you call `check_all_locks()` it's like running a 🔧random 🔧simulation.
- And your *random simulation* is happening *every time* the loop repeats, which is continuously.

This is a bug.

The simulation should only run *once* when the hatch is closed, and not continuously.

- You will fix this in the next Objective.
- Can your code keep track of whether the microswitch is *currently pressed?*

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  from random import randint
4
5  # Set up the peripherals
6  power.enable_periph_vcc(True)
7  np = neopixel.NeoPixel(exp.PORT0, 8)
8  microswitch = exp.digital_in(exp.PORT2)
9
10 # Constants for the peripherals
11 HATCH_CLOSED = False    # Microswitch
12 HATCH_OPEN = True       # Microswitch
13
14 # Brightness variables and constants.
15 RGB_RED = (20, 0, 0)
16 RGB_YELLOW = (20, 20, 0)
17 RGB_GREEN = (0, 20, 0)
18
19 # Set all pixels a random color
20 def disco_ball():
21     for pixel in range(8):
22         red = randint(0, 20)
23         green = randint(0, 20)
```

```
24        blue =randint(0, 20)
25        rgb_color = (red, green, blue)
26        np[pixel] = rgb_color
27
28    # Simulate if a lock was successful or failed
29    def is_lock_failed():
30        return randint(0, 99) < 15 # 15% failure
```

Define a function that will return `True` 15% of the time.

- Generate a random number between 0 and 99.
- Compare the number to 15.
- If less than 15, return `True` (or fail)
- Otherwise return `False` (or pass)

```
31
32    # Check each lock one at a time
33    def check_all_locks():
34        for pixel in range(8):
```

Define a function that checks the status of each lock individually.

- Use a `for` loop to check each lock one at a time.
- You have 8 locks, so `range(8)`.

```
35            if is_lock_failed():
36                np[pixel] = RGB_RED
37            else:
38                np[pixel] = RGB_GREEN
```

If the lock fails, set the individual pixel **RED**.
Else, set the individual pixel **GREEN**.

```
39
40    # Set all pixels to one color
41    def set_all_pixels(rgb_color):
42        for pixel in range(8):
43            np[pixel] = rgb_color
44
45    while True:
46        # Read the microswitch and check each lock
47        if microswitch.value == HATCH_CLOSED:
48            check_all_locks()
```

Call `check_all_locks()` if the **hatch** is **CLOSED**.

```
49        else:
50            set_all_pixels(RGB_RED)
```

Hints:

- ## Do the math!

  According to a local statistician:

  - All 8 locks will successfully engage only 27.2% of the time.

- ## 15% Fail Rate

  What is a 15% failure rate?

  - If you have 100 locks, 15 fail.

  Simulate the failure rate by generating a random number between 0 and 9 9. Use the numbers between 0 and 14 to represent the 15% failure rate. The numbers between 15 and 99 represent the 85% success rate.

  - Use a **condition** to compare the random number to 1 5.

- If less than 15, return `True` (or fail).
- Otherwise return `False` (or pass).

The code will look like this:

```
return randint(0, 99) < 15
```

Goals:

- Define the 🔧function `is_lock_failed()`.
- `return True` 15% of the time using `randint(0, 99) < 15`.
- Define the 🔧function `check_all_locks()`.
- Use `is_lock_failed()` as the 🔧condition to an `if` statement.
- Call `check_all_locks()`.

Tools Found: Functions, Random Numbers, Computer Simulations, bool

Solution:

```
1  from codex import *
2  from time import sleep
3  from random import randint
4
5  # Set up the peripherals
6  power.enable_periph_vcc(True)
7  np = neopixel.NeoPixel(exp.PORT0, 8)
8  microswitch = exp.digital_in(exp.PORT2)
9
10 # Constants for the peripherals
11 HATCH_CLOSED = False    # Microswitch
12 HATCH_OPEN = True       # Microswitch
13
14 # Brightness variables and constants.
15 RGB_RED = (20, 0, 0)
16 RGB_YELLOW = (20, 20, 0)
17 RGB_GREEN = (0, 20, 0)
18
19 # Set all pixels a random color
20 def disco_ball():
21     for pixel in range(8):
22         red = randint(0, 20)
23         green = randint(0, 20)
24         blue =randint(0, 20)
25         rgb_color = (red, green, blue)
26         np[pixel] = rgb_color
27
28 # Simulate if a lock was successful or failed
29 def is_lock_failed():
30     return randint(0, 99) < 15 # 15% failure
31
32 # Check each lock one at a time
33 def check_all_locks():
34     for pixel in range(8):
35         if is_lock_failed():
36             np[pixel] = RGB_RED
37         else:
38             np[pixel] = RGB_GREEN
39
40 # Set all pixels to one color
41 def set_all_pixels(rgb_color):
42     for pixel in range(8):
43         np[pixel] = rgb_color
44
45 while True:
46     # Read the microswitch and check each lock
47     if microswitch.value == HATCH_CLOSED:
```

```
48          check_all_locks()
49      else:
50          set_all_pixels(RGB_RED)
```

Objective 8 - Was the Button Pressed?

Track the *state* of your microswitch.

Sometimes your code needs to have ***memory.***

You want to check the locks only when the microswitch is first pressed, and not continuously after that.

- Right now, it doesn't remember whether the *microswitch* was just pressed or not.
- You can give your code *memory* with a variable.

Use a *variable* to track whether the microswitch *was pressed*.

- The initial value will be `False` (it has not yet been pressed).

```
was_pressed = False
```

- Update this variable inside your loop when the microswitch *changes.*

Check the 'Trek!

Follow the CodeTrek to add a variable to remember the microswitch pressed status.

Physical Interaction: *Try it!*

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  from random import randint
4
5  # Set up the peripherals
6  power.enable_periph_vcc(True)
7  np = neopixel.NeoPixel(exp.PORT0, 8)
8  microswitch = exp.digital_in(exp.PORT2)
9
10 # Constants for the peripherals
11 HATCH_CLOSED = False    # Microswitch
12 HATCH_OPEN = True       # Microswitch
13
14 # Brightness variables and constants.
15 RGB_RED = (20, 0, 0)
16 RGB_YELLOW = (20, 20, 0)
17 RGB_GREEN = (0, 20, 0)
18
19 # Set all pixels a random color
20 def disco_ball():
21     for pixel in range(8):
22         red = randint(0, 20)
23         green = randint(0, 20)
24         blue =randint(0, 20)
25         rgb_color = (red, green, blue)
26         np[pixel] = rgb_color
27
28 # Simulate if a lock was successful or failed
29 def is_lock_failed():
30     return randint(0, 99) < 15 # 15% failure
31
32 # Check each lock one at a time
33 def check_all_locks():
34     for pixel in range(8):
35         if is_lock_failed():
36             np[pixel] = RGB_RED
37         else:
```

```
38            np[pixel] = RGB_GREEN
39
40  # Set all pixels to one color
41  def set_all_pixels(rgb_color):
42      for pixel in range(8):
43          np[pixel] = rgb_color
44
45  # Main program
46  was_pressed = False
```

Define a variable that keeps track of the microswitch status.

- The variable needs to be defined above the `while True:` loop.
- The initial value is `False`, or not pressed.
- Update the variable every time the status changes.

```
47
48  while True:
49      # Read the microswitch and check each lock
50      if microswitch.value == HATCH_CLOSED and was_pressed == False:
```

Modify the `if` statement to be `True` only when:

- the **hatch** is **closed** ***AND***
- the microswitch hasn't been pressed.

```
51          was_pressed = True
```

Update `was_pressed` to `True`.

- The microswitch status changed.

```
52          check_all_locks()
53      elif microswitch.value == HATCH_OPEN:
```

Change `else` to `elif`.

- You only want the pixels `RED` if the **hatch** is **OPEN**.

```
54          was_pressed = False
```

Once the **hatch** opens, update `was_pressed` to `False`.

- The microswitch status changed.

```
55          set_all_pixels(RGB_RED)
```

Hints:

- ## What about the disco ball?

  The disco ball was pretty fun! And you still have the code. After you finish the mission and you can indicate when the hatch lock is closed, you can add back in the disco ball.

  This is optional!

  If you want help in adding code for the disco ball, go to the next hint.

- ## Turn on/off the disco ball

  You will need a mechanical device to turn on/off the disco ball. Let's use the switch.

  Set up the **switch** on **PORT3**

```
switch = exp.digital_in(exp.PORT3)
```

The switch will need a constant for POWER_ON.

```
POWER_ON = False
```

- ## Modify the if statement

Now you have everything you need to entertain the crew with a disco ball.

Modify the `if` statement in the `while True:` loop to include the disco ball as a branch.

The disco ball is the most restrictive condition, so it will go first.

```
while True:
    # Read the microswitch and switch and display the NeoPixels
    if switch.value == POWER_ON:
        disco_ball()
        sleep(0.2)
    elif microswitch.value == HATCH_CLOSED and was_pressed == False:
        was_pressed = True
        check_all_locks()
    elif microswitch.value == HATCH_OPEN:
        was_pressed = False
        set_all_pixels(RGB_RED)
```

Goals:

- Use an `if` statement with the 🔧condition `microswitch.value == HATCH_CLOSED and was_pressed == False`.
- Use an `elif` statement with the 🔧condition `microswitch.value == HATCH_OPEN`.
- Assign `was_pressed` as `True`.
- Assign `was_pressed` as `False` at least two times.

Tools Found: Variables, Loops, bool, undefined

Solution:

```
1  from codex import *
2  from time import sleep
3  from random import randint
4
5  # Set up the peripherals
6  power.enable_periph_vcc(True)
7  np = neopixel.NeoPixel(exp.PORT0, 8)
8  microswitch = exp.digital_in(exp.PORT2)
9
10 # Constants for the peripherals
11 HATCH_CLOSED = False   # Microswitch
12 HATCH_OPEN = True      # Microswitch
13
14 # Brightness variables and constants.
15 RGB_RED = (20, 0, 0)
16 RGB_YELLOW = (20, 20, 0)
17 RGB_GREEN = (0, 20, 0)
18
19 # Set all pixels a random color
20 def disco_ball():
21     for pixel in range(8):
22         red = randint(0, 20)
23         green = randint(0, 20)
24         blue =randint(0, 20)
25         rgb_color = (red, green, blue)
26         np[pixel] = rgb_color
27
28 # Simulate if a lock was successful or failed
29 def is_lock_failed():
30     return randint(0, 99) < 15 # 15% failure
31
32 # Check each lock one at a time
```

```
33  def check_all_locks():
34      for pixel in range(8):
35          if is_lock_failed():
36              np[pixel] = RGB_RED
37          else:
38              np[pixel] = RGB_GREEN
39
40  # Set all pixels to one color
41  def set_all_pixels(rgb_color):
42      for pixel in range(8):
43          np[pixel] = rgb_color
44
45  # Main program
46  was_pressed = False
47
48  while True:
49      # Read the microswitch and check each lock
50      if microswitch.value == HATCH_CLOSED and was_pressed == False:
51          was_pressed = True
52          check_all_locks()
53      elif microswitch.value == HATCH_OPEN:
54          was_pressed = False
55          set_all_pixels(RGB_RED)
```

Mission 4 Complete

You've completed project Hatch Lock!

You now know all sorts of information about NeoPixels. You have also given the crew a safe, secure way to determine the position of the hatch locks!

Great work!!

Mission 5 - Alert System!

This project shows you how to read multiple analog sensors and display alerts to the crew!

Mission Briefing:

Time is of the essence if the ship has technical problems. The crew will need an alerting system to tell them if something has gone wrong.

A risk analysis determined that two potential dangers exist and must be monitored:

1. There are multiple electronic devices operating in the ship's electronics bay. The crew must be notified immediately if the **temperature** in the electronics bay exceeds a **normal** operating range.
2. Sections of the ship must be cordoned off in the event of a hull breach. The crew must be notified immediately if an **explosion** is detected.

Project: Alert System will guide you through informing the crew if there is an emergency.

Project Goals:

- Use multiple analog sensors
- Learn to convert temperature values
- Use an average calculation
- Alert the crew if you detect an emergency situation!!

Objective 1 - Red Light

The first step is preparing our system to display an alert.

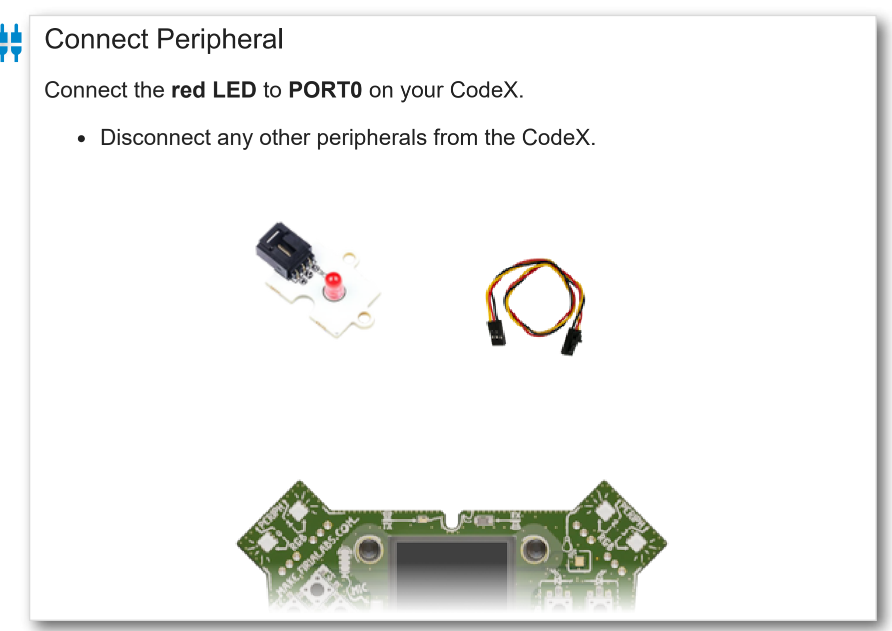
- The red LED will be the alerting mechanism.
- When the LED is **ON** the crew needs to take immediate action.
- You will add **sensors** later.

First step is to get the LED working!

Connect Peripheral

Connect the **red LED** to **PORT0** on your CodeX.

- Disconnect any other peripherals from the CodeX.



Coding a *Sensor / Alert* framework

It would be easy to just turn **ON** the LED.

- But you know you'll soon need the following loop:

- Read sensors
- Check sensor values
- Set alert output
- *Repeat!*

Create a New File!

Use the **File** → **New File** menu to create a new file called ***AlertSystem***.

Check the 'Trek!

Follow the CodeTrek to code your basic Sensor / Alert framework.

Run It!

Make sure your LED is always turned **ON**.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  # TODO: Define LED_ON
6  # TODO: Define LED_OFF
```

Start the program with the libraries and constants you will need.

- Import from **codex** and **time**.
- Define constants for `LED_ON` and `LED_OFF`.

```
7
8  # Set up the peripherals
9  led = exp.digital_out(exp.PORT0)
```

Set up the LED on **PORT0**.

- You will use only one LED for this mission, so you don't have to specifiy white or red.

```
10
11 def set_led(val):
12     # TODO: Set the led value
```

Define a function for turning on/off the LED.

- Use a parameter `val`.
- `led.value = val`

```
13
14 # Main Program
15 while True:
16     # TODO: Call set_led() to turn ON the red LED
17     sleep(1)
```

Use a `while True:` loop to call the LED function.

- `set_led(LED_ON)`
- The red light should stay **ON**.

Hints:

- ## Familiar Start

  This program starts the same as the first two missions:

  - Import the libraries.
  - Set up the **LED** on **PORT0**
  - Set up 🔧constants for `LED_ON` and `LED_OFF`.
  - Define a 🔧function for turning on/off the LED.
  - Call the function in a `while True:` loop.

  How much of this can you do on your own?

- ## Save Some For Later

  During this Objective, your loop will only set the alert output (turn the red LED **ON**). You will continue to modify the loop as you work through the Objectives.

Goals:

- Create a new file named `AlertSystem`.
- Define 🔧constants `LED_ON` and `LED_OFF`.
- Define the 🔧function `set_led(val)`.
- Assign the value of `led.value` as `val`.
- Call `set_led()`.

Tools Found: Loops, Constants, Functions

Solution:

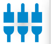
```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = True
6  LED_OFF = False
7
8  # Set up the peripherals
9  led = exp.digital_out(exp.PORT0)
10
11 def set_led(val):
12     led.value = val
13
14 # Main Program
15 while True:
16     set_led(LED_ON)
17     sleep(1)
```

Objective 2 - Blinking Red Alert

The next step is simulating a sensor for testing purposes.

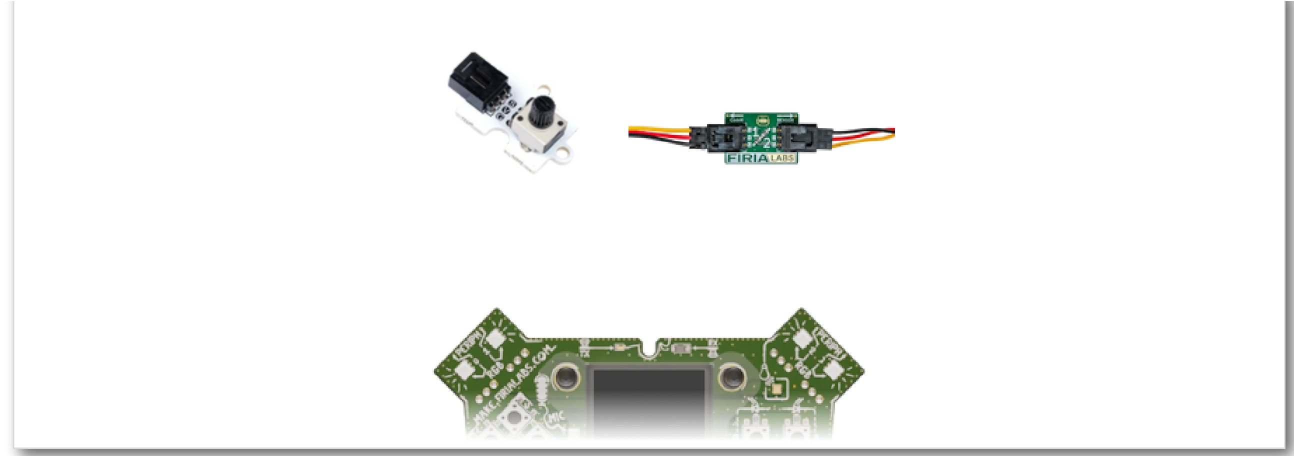
The potentiometer will simulate the first sensor.

- If the knob is rotated past half way it is time to alert the crew!
- The potentionmeter will use the resistor voltage divider to limit the ADC input.



Connect Peripheral

Connect the **divider** to **PORT1** and the **potentiometer** to the **divider** on your CodeX.



Wouldn't it be more authentic if the light *blinks* when it is on?

- Use what you learned about 🔧PWM to *blink* the red LED when it is **ON**.
- You need to set the **Frequency** and **Duty-Cycle**.

Duty Cycle
ON
OFF
Analog Period

Frequency (Analog Period)

To set your LED to **blink** two times per second, change the LED set up to:

```
led = exp.pwm_out(exp.PORT0, frequency=2)
```

Duty-Cycle

For now, you will use the potentiometer to control the LED.

- Set the duty_cycle to half the highest value of the potentiometer.
- Check the 💡 Hints for how the duty_cycle is calculated.
- Change the `LED_ON` constant and the `LED_OFF` constant to integer values:

```
LED_ON = 2**15 # duty_cycle for LED
LED_OFF = 0
```

The constants will be passed to the parameter for in `set_led()`:

```
def set_led(val):
    led.duty_cycle = val
```

Check the 'Trek!

Follow the CodeTrek to:

- Set up the potentiometer on PORT1.
- Modify your `set_led(val)` function to assign the duty_cycle to the LED.
- Modify the `while True:` loop so the potentiometer controls the LED.

Physical Interaction: *Try it!*

Twist the knob back and forth and watch the red LED.

- The LED should **blink** when the knob is turned up.
- The LED should be **OFF** when the knob is turned down.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
```

```
4  # Constants for peripherals
5  LED_ON = 2**15    # Half-power duty_cycle
6  LED_OFF = 0
```

Change the value of the constants as integers for the duty_cycle.

- `LED_ON` is assigned the value to make it blink.
  - It is halfway between 0 and 2^{16}
  - Or 2^{15}

```
7
8  # Set up the peripherals
9  led = exp.pwm_out(exp.PORT0, frequency=2)
```

Modify the LED set up to use PWM with a frequency of 2.

```
10 potentiometer = exp.analog_in(exp.PORT1)
```

Set up the potentiometer on PORT1.

- Remember: It is an analog input peripheral.

```
11
12 def set_led(val):
13     led.duty_cycle = val
```

Modify `set_led()` to set the duty_cycle.

- The **ON** duty-cycle is `LED_ON`.
- The **OFF** duty-cycle is `LED_OFF`.

```
14
15 # Main Program
16 while True:
17     if potentiometer.value > LED_ON:
18         set_led(LED_ON)
19     else:
20         set_led(LED_OFF)
```

Modify the `while True:` loop so that the potentiometer controls the LED.

Hints:

- ## Attention! Analog Device

  The potentiometer is an analog input sensor.

- Set up the potentiometer with `exp.analog_in(exp.PORT1)`

- ## Potentiometer Particulars

- The value returned by the knob is between 0 and 65535 (or 2^{16}).
- Half the highest value of the potentiometer is 65536/2 or 3276 8.
- This is the same value as 2^{15}.

- So if the `value > 32768` → the knob has crossed the half-way point.

Goals:

- Set up the **LED** by assigning the variable `led` as the output of `exp.pwm_out(exp.PORT0, frequency=2)`.
- In `set_led()`, assign `led.duty_cycle` as `val`.
- In an `if` statement, use the condition `potentiometer.value > BLINKING`.

- Call `set_led()` at least twice.

Tools Found: PWM, Variables, bool

Solution:

```
from codex import *
from time import sleep

# Constants for peripherals
LED_ON = 2**15 # Half-power duty_cycle
LED_OFF = 0

# Set up the peripherals
led = exp.pwm_out(exp.PORT0, frequency=2)
potentiometer = exp.analog_in(exp.PORT1)

def set_led(val):
    led.duty_cycle = val

# Main Program
while True:
    if potentiometer.value > LED_ON:
        set_led(LED_ON)
    else:
        set_led(LED_OFF)
```

Objective 3 - Show me the Values!

Streaming values from your sensor

*Wouldn't it be nice if there was a way to **continuously display sensor values to a screen?***

Python's built-in `print()` function is made for that!

- You'll need to open the **Console Panel** in *CodeSpace* to use this feature.
    - Use the ☰ button.
    - The Console Panel button is below the *toolbox button*.
- The **Console Panel** will open below the Instructions Panel.
- The **Console Panel** lets you see the output of a `print()` statement.
- The **Console Panel** also lets you type in Python statements directly. This is called the **REPL**.

Concept: *REPL*

"Read Evaluate Print Loop"

REPL is a name for the "command line" that languages like **Python** offer.

Besides being a place to see `print()` statement *output*, the REPL is a great way to test out snippets of code, language features, and APIs as you decide how to use them in your code.

The `print()` statement can do a lot more than just display text. You can give it multiple arguments:

- strings
- integers
- lists
- etc.

Example: `print("Your name is", name, "and your address is", address)`

Check the Hints for more details about the Console Panel and `print()`.

Check the 'Trek!

Follow the CodeTrek to modify your code to `print()` a **label** and your knob's analog **value** each time the system loops.

- The output should look something like: ***"Knob value: 28536"***

Physical Interaction: *Try it!*

Run the code. Rotate the knob back and forth and watch the values in the Console Panel.

- Now you can see *exactly* what the value of your knob is!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**15     # Half-power duty cycle
6  LED_OFF = 0
7
8  # Set up the peripherals
9  led = exp.pwm_out(exp.PORT0, frequency=2)
10 potentiometer = exp.analog_in(exp.PORT1)
11
12 def set_led(val):
13     led.duty_cycle = val
14
15 # Main Program
16 while True:
17     print("Knob Value: ", potentiometer.value)
```

Add a `print()` statement in the `while True:` loop to display the knob value.

```
18
19     if potentiometer.value > LED_ON:
20         set_led(LED_ON)
21         print("Alert, alert, alert!")
```

Optional

You can also display an **Alert** message with the blinking light.

```
22     else:
23         set_led(LED_OFF)
```

Hints:

- ## The Console Panel

  When you open the **Console Panel**, text will already show.

  - Ignore any text that is there for now.
  - Those messages are a side effect of stopping and starting your code.

  You should see `>>>_` with the cursor *blinking*.

  - The output from `print()` statements will display there.

- ## The Console Panel

  Your code will display output in real time, which is really fast.

  - That means the output will display so quickly you won't be able to read it.
  - After you stop the code, you can scroll back through the display to see the output.

- ## Print Statement Particulars

When you add arguments to a `print()` statement, you will separate the arguments with a comma (,).

- A **string** argument must be in quotation marks (" ").
- A **variable** argument is NOT in quotation marks.

- ## Print Statement Example

  This example has four arguments:

  - Two string arguments
  - Two variable arguments

```
print("Your name is", name, "and your address is", address)
```

Goals:

- Click the ≡ button at the lower-right to open the **Console Panel**.
- Print `potentiometer.value`.

Tools Found: Print Function, REPL, Keyword and Positional Arguments, str, int, list, Analog to Digital Conversion

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**15     # Half-power duty cycle
6  LED_OFF = 0
7
8  # Set up the peripherals
9  led = exp.pwm_out(exp.PORT0, frequency=2)
10 potentiometer = exp.analog_in(exp.PORT1)
11
12 def set_led(val):
13     led.duty_cycle = val
14
15 # Main Program
16 while True:
17     print("Knob Value: ", potentiometer.value)
18
19     if potentiometer.value > LED_ON:
20         set_led(LED_ON)
21         print("Alert, alert, alert!")
22     else:
23         set_led(LED_OFF)
24
```

Objective 4 - Raw Temperature!

The first danger the crew should be alerted to is a high temperature in the electronics bay.

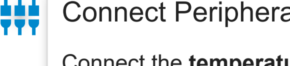
High temperatures can lead to many potential issues:

- Electronics could be irreversibly damaged.
- The devices could catch fire.
- Certain devices have the potential for explosion.

Modify your alerting system to monitor **temperature** in the ship's electronics bay.

- Then you can alert the crew if the temperature exceeds a **threshold**!
- Use the **temperature sensor** from the peripherals kit to monitor the temperature.
- The temperature sensor is an analog input sensor **just like the potentiometer!!**, so your code won't have to change much for this :-)

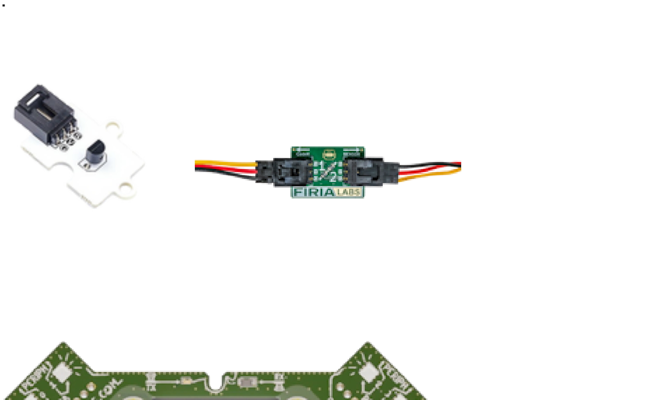
- Like the potentiometer and other analog input devices, the temperature sensor will use the resistor voltage divider to limit the ADC input.



Connect Peripheral

Connect the **temperature sensor** to the **divider** on **PORT1** on your CodeX.

- First disconnect the **potentiometer**.



Know Your *Sensor*

The temperature sensor outputs exactly **0.75 Volts at 25°C**

- The output changes by **± 0.01 Volts per degree C**

Wait! ***Volts?***

This sensor's output is given in *volts*, but `temp_sensor.value` gives a **raw** ADC *value* from 0-65535, just like the Potentiometer.

- You will need to convert the **raw data** to *real units*: Volts → °C.

Raw analog values won't mean much to the crew.

You'll need to write some *conversion code!*

- The sensor manufacturer provided the calculation to do this.
- Check the Hints for details about the temperature sensor and how it works.

Check the 'Trek!

Follow the CodeTrek to set up the temperature sensor and convert the raw temperature data to Celsius.

Run It!

Open the **Console Panel** and watch the temperature values.

- Squeeze the sensor between your fingers to change the temperature.
- Can you get temperatures that display the **Alert** message?
- Can you get temperatures that turn off the message?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**15       # Half-power duty cycle
6  LED_OFF = 0
```

```
7
8
9  # Set up the peripherals
10 led = exp.pwm_out(exp.PORT0, frequency=2)
11 temp_sensor = exp.analog_in(exp.PORT1)
```

Set up the temperature sensor on **PORT1**.

- Change the line of code for setting up the potentiometer to the temperature sensor.

```
12
13 def set_led(val):
14     led.duty_cycle = val
15
16 def convert_temp_to_c(raw_temp):
17     # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
18     degrees_c = raw_temp * 0.004577 - 50
19     return degrees_c
```

Define a function for converting the raw temperature value to celsius.

- Check the Hints for detailed information on the calculation.
- The function returns the temperature reading in Celsius.
- You do not need to type in the `# comment`.

```
20
21 # Main Program
22 while True:
23     degree_c = convert_temp_to_c(temp_sensor.value)
24     print("Celsius: ", degree_c)
```

Define a variable `degrees_c` and assign it the temperature reading in Celsius.

Then `print()` the converted temperature to the Console Panel.

```
25
26     if degree_c > 26:
```

Modify the `if` statement to compare `degree_c`.

- Also change the value being compared from `LED_ON` to `26`.
- The value 26 is a suggestion. You can try different values.

```
27         set_led(LED_ON)
28         print("Alert, alert, alert!")
29     else:
30         set_led(LED_OFF)
```

Hints:

- ## Know Your *Sensor*

  The temperature sensor is quite **accurate**.

  - It is typically accurate within **± 2° Celsius** over a wide range (-40°C to +125°C).
  - It's also *pre-calibrated at the factory*.
  - It outputs exactly **0.75 Volts at 25°C**
  - The output changes by **± 0.01 Volts per degree C**

- ## Convert Raw Data to Celsius

  How can you convert a (0 - 65535) 🔧ADC value to °C?

  The conversion can be split into two parts:

  1. Convert ADC counts to voltage.

2. Convert voltage to temperature.

Using Ohm's Law:

Voltage = Current * Resistance

- The change in temperature changes the resistance.
- The change in resistance causes a change in voltage.
- The **temperature sensor** measures the voltage to "read" the temperature.
- The material in the temperature sensor has a near linear resistance/temperature gradient.
- So use the **slope/intercept** formula to convert the raw data to Celsius:
- y = mx + b

Try it out!

- The temperature sensor outputs exactly 0.75 Volts at 25°C
- Calculate **b** given:

y = 25, m = 100, x = 0.75

- ## Convert Volts to Celsius

This calculation was provided by the sensor manufacturer. It converts the raw temperature values to degrees Celsius.

- °C = $\frac{((rawtemp/65535)*3000mV - 500mV)}{10deg_p er_m V}$
- Simplifying the math:
- °C = $\frac{(rawtemp*0.045777)}{10} - \frac{500}{10}$
- °C = $rawtemp * 0.0045777 - 50$

Use this calculation in a function to convert to celsius.

- ## Change It Up

The temperature sensor notices even small changes. When running the code, try:

- Placing something very cool close to the sensor and get the temperature to drop.
- Squeezing the sensor between your fingers to warm it up and get the temperature to rise.

Goals:

- Define the 🔧function `convert_temp_to_c(raw_temp)`.
- Assign the 🔧variable `degress_c` as the output of `convert_temp_to_c()`.
- 🔧Print `degree_c`.
- Use an `if` statement with the 🔧condition `degress_c > 26`

Tools Found: Analog to Digital Conversion, Functions, Variables, Print Function, bool

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**15      # Half-power duty cycle
6  LED_OFF = 0
7
8  # Set up the peripherals
```

```
 9  led = exp.pwm_out(exp.PORT0, frequency=2)
10  temp_sensor = exp.analog_in(exp.PORT1)
11
12  def set_led(val):
13      led.duty_cycle = val
14
15  def convert_temp_to_c(raw_temp):
16      # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
17      degrees_c = raw_temp * 0.004577 - 50
18      return degrees_c
19
20  # Main Program
21  while True:
22      degree_c = convert_temp_to_c(temp_sensor.value)
23      print("Celsius: ", degree_c)
24
25      if degree_c > 26:
26          set_led(LED_ON)
27          print("Alert, alert, alert!")
28      else:
29          set_led(LED_OFF)
```

Quiz 1 - Check Your Understanding: Temp Alert

Question 1: What are two properties that need to be set when using **PWM**?

✓ Frequency and duty-cycle

✗ Wavelength and amplitude

✗ Height and depth

✗ Time and distance

Question 2: What is the correct way to `print()` a string and a variable?

✓ `print("My string", variable)`

✗ `print(My string, variable)`

✗ `print("My string", "variable")`

✗ `print("My string"  variable)`

Question 3: **REPL** can be used for many things. Which of the following can it **NOT** be used for?

✓ Changing the sensor PORT.

✗ See `print()` statement output.

✗ Test snippets of code.

✗ Test language features and APIs.

Question 4: What is output by the temperature sensor?

✓ Volts

✗ Temperature in degrees

✗ Temperature in celsius

✗ Sound waves

Objective 5 - It's Getting Hot in Here!

Now finish the temperature alerting system.

You can see the current temperature in the electronics bay, and you need to decide when the alarm should be triggered.

- *How hot does the bay need to get before the crew is alerted?*
- This rise in temperature is the **threshold**.
- You will simulate a rise in temperature with your fingers, so we will keep the threshold small for now.

Here is the algorithm for completing the temperature alerting system.

- Define a constant for `TEMP_THRESHOLD`.
- Read the temperature sensor value `first_temp` to get the room temperature.
- Calculate the `temp_limit` that determines when the alarm is triggered.
- Use `temp_limit` in the `if` statement's condition.
- If triggered, display an alert message, turn on the LED, and break out of the loop so the crew can reset the system after they handle the emergency.

Check the 'Trek!

Follow the CodeTrek to finish setting up the temperature alerting system by following the algorithm.

Physical Interaction: *Try it!*

Run the code. Squeeze the temp sensor between your fingers until the temperature rises above your `TEMP_THRESHOLD`!

- Make sure the **Console Panel** is open and you are watching the temperature values.
- When the temperature is above the threshold, the first loop will break and the LED will continue to blink until the code is stopped.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**15       # Half-power duty cycle
6  LED_OFF = 0
7  TEMP_THRESHOLD = 2  # Degrees Celsius
```

Define a constant `TEMP_THRESHOLD` and set it to `2`.

- The constant is for 2 Degrees Celsius above normal temperature.
- This will allow you to simulate a high temperature to trigger an alert.
- It should trigger the alarm when the temperature is 2 degrees above the sensor's first reading.

```
8
9  # Set up the peripherals
10 led = exp.pwm_out(exp.PORT0, frequency=2)
11 temp_sensor = exp.analog_in(exp.PORT1)
12
13 def set_led(val):
14     led.duty_cycle = val
15
16 def convert_temp_to_c(raw_temp):
17     # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
18     degrees_c = raw_temp * 0.004577 - 50
19     return degrees_c
20
21 def set_alarm():
22     global temp_limit
23     first_temp = convert_temp_to_c(temp_sensor.value)
```

Define a function `set_alarm()` that reads the initial temperature and converts it to celsius.

- The function will call `convert_temp_to_c()` to convert the temperature sensor's **raw temp** to **Celsius**.
- The initial Celsius temperature is assigned to `first_temp`.

```
24     # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
25     temp_limit = first_temp + TEMP_THRESHOLD
```

Assign the actual temperature threshold to `temp_limit`.

- This is the temperature value that will trigger the alarm.
- For this simulation, it will be 2 degrees above `first_temp`.
- `temp_limit` is a global variable, used in the **Main Program**.
  - Remember to make it **global**.

```
26
27 # Main Program
28 set_alarm()
```

Call `set_alarm()` in the **Main Program** before the `while True:` loop.

- This function sets the `temp_limit`, so it needs to executed before the rest of the code.

```
29 while True:
30     degree_c = convert_temp_to_c(temp_sensor.value)
31     print("Celsius: ", degree_c)
32
33     if degree_c > temp_limit:
```

Use `temp_limit` in the `if` statement to trigger the alarm.

```
34         print("High Temperature Alert!")
35         set_led(LED_ON)
36         break
```

Add a `print()` statement to display a message about the danger.

- Add a `break` to stop the loop.
- *The crew will need to reset the system after they handle the emergency!*

```
37     else:
38         set_led(LED_OFF)
39
40 # After the loop breaks, keep the LED flashing
41 while True:
42     set_led(LED_ON)
```

Add another `while True:` loop that keeps the LED blinking.

- This loop will only run after the `break` in the first loop.
- It will keep the LED blinking until the crew fixes the problem and resets the alarm.

Hint:

- ## Temp Threshold

  A **threshold** of 2 degrees isn't very realistic because change in temperature that small probably shouldn't cause an alarm.

  - To get the actual **threshold** your team of engineers would do some testing and decide how much hotter than "normal" the bay should be to cause alarm.
  - For this simulation, 2 degrees is a good threshold, or value limit, for testing purposes.

Goals:

- Define the 🔧function `set_alarm()`.
- In it, assign the 🔧variable `first_temp`.
- Call `set_alarm()`.
- In an `if` statement, use the 🔧condition `degree_c > temp_limit`.
- 🔧Print `"High Temperature Alert!"`.

Tools Found: Algorithm, Functions, Variables, bool, Print Function

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**15        # Half-power duty cycle
6  LED_OFF = 0
7  TEMP_THRESHOLD = 2    # Degrees Celsius
8
9  # Set up the peripherals
10 led = exp.pwm_out(exp.PORT0, frequency=2)
11 temp_sensor = exp.analog_in(exp.PORT1)
12
13 def set_led(val):
14     led.duty_cycle = val
15
16 def convert_temp_to_c(raw_temp):
17     # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
18     degrees_c = raw_temp * 0.004577 - 50
19     return degrees_c
20
21 def set_alarm():
22     global temp_limit
23     first_temp = convert_temp_to_c(temp_sensor.value)
24     # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
25     temp_limit = first_temp + TEMP_THRESHOLD
26
27 # Main Program
28 set_alarm()
29 while True:
30     degree_c = convert_temp_to_c(temp_sensor.value)
31     print("Celsius: ", degree_c)
32     if degree_c > temp_limit:
33         print("High Temperature Alert!")
34         set_led(LED_ON)
35         break
36     else:
37         set_led(LED_OFF)
38
39 # After the loop breaks, keep the LED flashing
40 while True:
41     set_led(LED_ON)
```

Objective 6 - Explosion Detection?

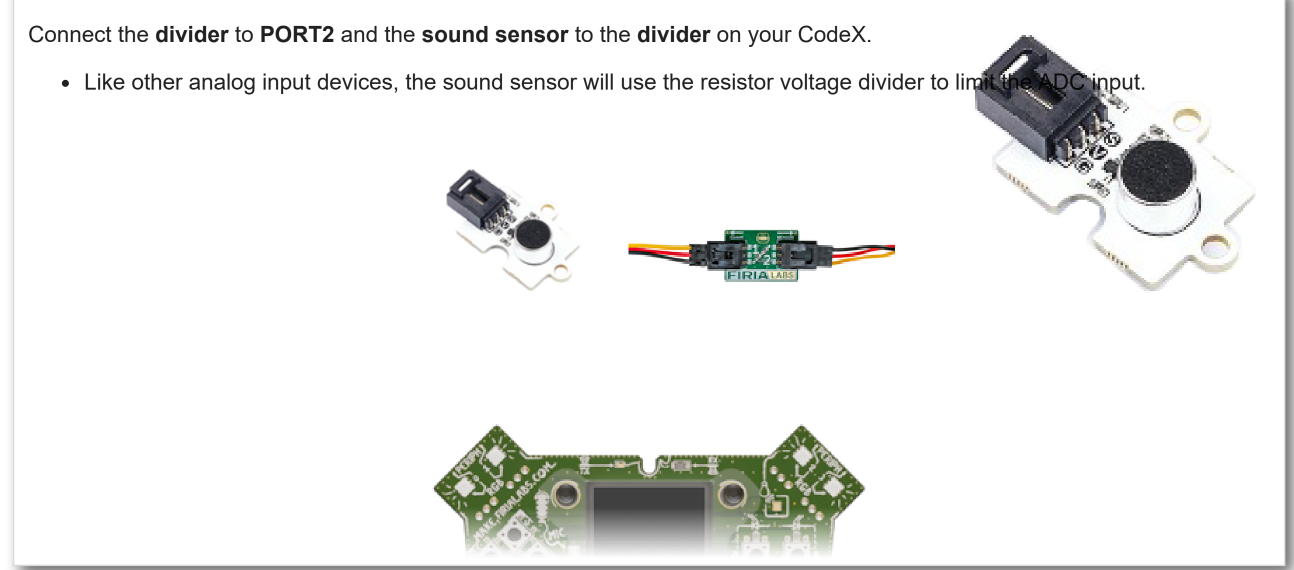
A second potential for danger aboard the shuttle is the risk of explosion or hull breach.

But, what can you use to detect an explosion?

How about listening for loud sounds!

- The **sound sensor** is an **analog** sensor similar to the **temperature sensor**.
- Check the 💡 Hints for more details about the sound sensor.

Connect Peripheral



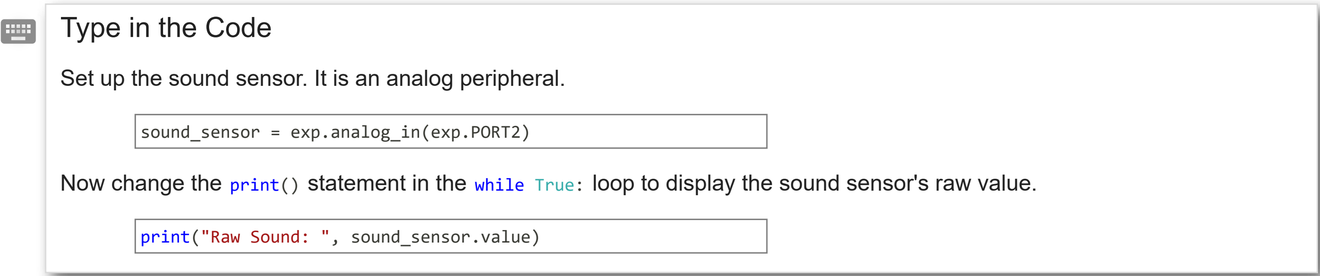
Connect the **divider** to **PORT2** and the **sound sensor** to the **divider** on your CodeX.

- Like other analog input devices, the sound sensor will use the resistor voltage divider to limit the ADC input.

The sound sensor is an analog sensor and the temperature sensor is an analog sensor.

They should work the same way... Right?

- Well, not exactly...
- The sound sensor only detects **momentary** changes in sound levels.



Type in the Code

Set up the sound sensor. It is an analog peripheral.

```
sound_sensor = exp.analog_in(exp.PORT2)
```

Now change the `print()` statement in the `while True:` loop to display the sound sensor's raw value.

```
print("Raw Sound: ", sound_sensor.value)
```

Run It!

Open the **Console Panel** and watch the output.

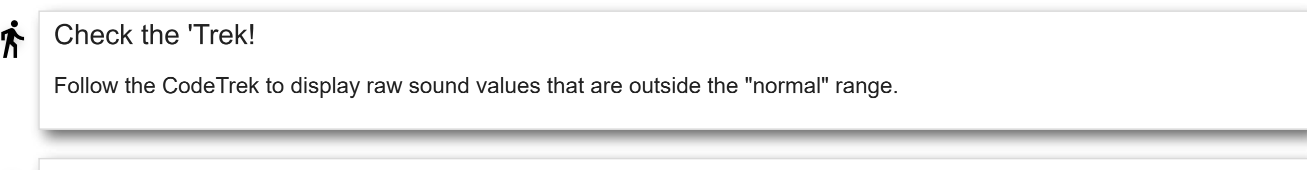
- Stop the program.
- Scroll back through the values and look at the range of raw sound values.

You should notice the values stay within a range of about 1000.

- In my environment, the raw sound values typically stay between 1580 and 1680.

This range of values is the "normal" sound of your environment.

- What are the values when a loud bang is detected? Let's find out.
- While testing the system, you just need to see the values that aren't in the "normal" range.



Check the 'Trek!

Follow the CodeTrek to display raw sound values that are outside the "normal" range.

Physical Interaction: *Try it!*

Run the code. Open the **Console Panel** and watch the output.

- Clap your hands close to the sound sensor.
- Did a raw sound value display?
- Clap several times. Then stop the program and look at the output.
- Do you see *high* values **and** *low* values?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**16 // 2   # Half-power duty cycle
6  LED_OFF = 0
7  TEMP_THRESHOLD = 2    # Degrees Celsius
8
9  # Set up the peripherals
10 led = exp.pwm_out(exp.PORT0, frequency=2)
11 temp_sensor = exp.analog_in(exp.PORT1)
12 sound_sensor = exp.analog_in(exp.PORT2)
```

Set up the sound sensor on PORT2.

- It is an analog input peripheral.
- The code is similar to the temperature sensor.

```
13
14 def set_led(val):
15     led.duty_cycle = val
16
17 def convert_temp_to_c(raw_temp):
18     # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
19     degrees_c = raw_temp * 0.004577 - 50
20     return degrees_c
21
22 def set_alarm():
23     global temp_limit
24     first_temp = convert_temp_to_c(temp_sensor.value)
25     # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
26     temp_limit = first_temp + TEMP_THRESHOLD
27
28 # Main Program
29 set_alarm()
30 while True:
31     degree_c = convert_temp_to_c(temp_sensor.value)
32     if sound_sensor.value > 1750:
```

Use an `if` statement to `print()` only the raw sound values that are outside the "normal" range.

- `1750` is used in this example as the top of the "normal" range.
- Use the high value from ***your*** normal range.

```
33         print("Raw Sound: ", sound_sensor.value)
```

Modify the `print()` statement to display the raw sound values that are above "normal".

```
34
35     if degree_c > temp_limit:
36         print("High Temperature")
37         set_led(LED_ON)
38         break
39     else:
40         set_led(LED_OFF)
41
42 # After the loop breaks, keep the LED flashing
43 while True:
44     set_led(LED_ON)
```

Hints:

- ## The Sound Sensor

  The **sound sensor** is essentially a small microphone.

  - It measures the **intensity** of the sound (whether the sound is quiet or loud).
  - This particular sensor is not accurate enough to analyze speech patterns.

- ## The Console Panel

  The `print()` output is displayed very quickly on the Console Panel. And there is a lot of data!

  - After each program run, you can close the **Console Panel** and then open it again to **reset** the output.
  - All the data is still there, but it is easier to see the output of each program run without getting overwhelmed.

Goals:

- Set up the **sound sensor** by assigning the 🔧variable `sound_sensor` as the output of `exp.analog_in(exp.PORT2)`.
- In an `if` statement, use `sound_sensor.value` in the 🔧condition.
- 🔧Print `sound_sensor.value`.

Tools Found: Variables, bool, Print Function

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**15        # Half-power duty cycle
6  LED_OFF = 0
7  TEMP_THRESHOLD = 2    # Degrees Celsius
8
9  # Set up the peripherals
10 led = exp.pwm_out(exp.PORT0, frequency=2)
11 temp_sensor = exp.analog_in(exp.PORT1)
12 sound_sensor = exp.analog_in(exp.PORT2)
13
14 def set_led(val):
15     led.duty_cycle = val
16
17 def convert_temp_to_c(raw_temp):
18     # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
19     degrees_c = raw_temp * 0.004577 - 50
20     return degrees_c
21
22 def set_alarm():
23     global temp_limit
24     first_temp = convert_temp_to_c(temp_sensor.value)
25     # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
26     temp_limit = first_temp + TEMP_THRESHOLD
27
28 # Main Program
29 set_alarm()
30 while True:
31     degree_c = convert_temp_to_c(temp_sensor.value)
32     if sound_sensor.value > 1750:
33         print("Raw sound: ", sound_sensor.value)
34
35     if degree_c > temp_limit:
36         print("High Temperature Alert!")
37         set_led(LED_ON)
38         break
39     else:
40         set_led(LED_OFF)
41
42 # After the loop breaks, keep the LED flashing
```

```
43 while True:
44     set_led(LED_ON)
```

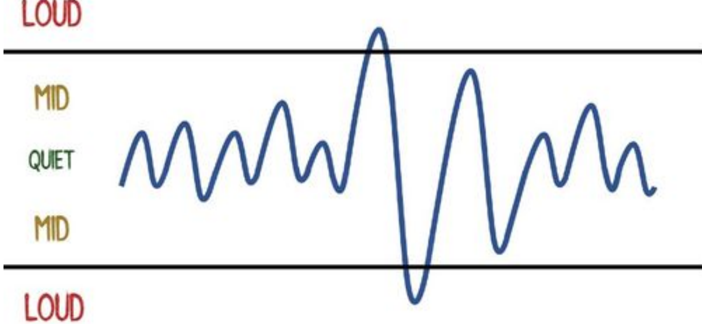
Objective 7 **- Sound Wave**

The sound sensor outputs raw values that mimic a sound wave!

What does that mean?

- A **louder** sound causes a **HIGHER** value **AND**
- A **louder** sound causes a **LOWER** value

Take a look at the following diagram:



The sound sensor outputs values similar to the diagram.

- Some raw values are high, and some are low.
- The average of the raw values is in the middle - not too high or too low.
- Check the Hints for more details about the sound sensor.

It is too unpredicable to determine the AVERAGE level of the sound sensor in advance.

- The sound sensor average values change based on:
  - Background noise
  - Temperature
  - Barometric pressure

So, how can I write code that automatically determines the average level for me?

There are **MANY** different ways to calculate an average, depending on the situation.

For this mission you will use the *Exponential Moving Average (EMA)* calculation.

- This calculation will put more **weight** on the newest value and less weight on the previous average.
- Check the Hints for more details about **EMA**.

Check the 'Trek!

Follow the CodeTrek to calculate the average sound and display the values.

Run It!

Open the **Console Panel** and watch the output.

- Stop the program.
- Scroll back through the values.
- Are the average sound values staying consistent?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**15       # Half-power duty cycle
6  LED_OFF = 0
7  TEMP_THRESHOLD = 2   # Degrees Celsius
8  WEIGHT = 0.02        # each new raw sound has a 2% significance
```

Define a constant `WEIGHT` to use in the EMA calculation for the sound average.

- The `WEIGHT` is 2%, or 0.02.

```
9
10 # Set up the peripherals
11 led = exp.pwm_out(exp.PORT0, frequency=2)
12 temp_sensor = exp.analog_in(exp.PORT1)
13 sound_sensor = exp.analog_in(exp.PORT2)
14
15 def set_led(val):
16     led.duty_cycle = val
17
18 def convert_temp_to_c(raw_temp):
19     # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
20     degrees_c = raw_temp * 0.004577 - 50
21     return degrees_c
22
23 # calculates average of raw sound values using EMA
24 def calc_avg(avg, new_val):
25     avg_sound = avg * (1 - WEIGHT) + new_val * WEIGHT
26     return avg_sound
```

Define the function `calc_avg()`, which performs the EMA calculation for the average sound level.

- It takes two parameters:
  - the current average `avg`.
  - the current new sound value `new_val`.
- It returns the new average `avg_sound`.

```
27
28 def set_alarm():
29     global temp_limit, avg_sound
30     first_temp = convert_temp_to_c(temp_sensor.value)
31     # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
32     temp_limit = first_temp + TEMP_THRESHOLD
33     # Initial sound reading
34     avg_sound = sound_sensor.value
```

Use the first sound value as the intial average.

- This is part of setting the alarm.
- `avg_sound` is a ***global*** variable.

```
35
36 # Main Program
37 set_alarm()
38 while True:
39     degree_c = convert_temp_to_c(temp_sensor.value)
40     avg_sound = calc_avg(avg_sound, sound_sensor.value)
41     print("Avg Sound: ", avg_sound)
```

Calculate a new `avg_sound` in the `while True:` loop.

- Call `calc_avg()` with the arguments `avg_sound` and `sound_sensor.value`.
- `print()` the `avg_sound`

```
42     sleep(0.5)
```

Add a half-second delay so the output isn't overwhelming.

```
43
44      if degree_c > temp_limit:
45          print("High Temperature Alert!")
46          set_led(LED_ON)
47          break
48      else:
49          set_led(LED_OFF)
50
51  # After the loop breaks, keep the LED flashing
52  while True:
53      set_led(LED_ON)
```

Hints:

- ## High and Low Values!?

  Did you notice both *high* and *low* raw sound values are displayed? Even though the `if` statement compared only a **high** value, some **low** values printed as well.

  Actual output from a test run:

```
Raw sound:  26829
Raw sound:  52586
Raw sound:  1846
Raw sound:  1688
Raw sound:  1429
Raw sound:  1906
Raw sound:  834
Raw sound:  1628
Raw sound:  52586
```

- ## Some NOTES about the sound sensor:
  - It will output values similar to the sound wave diagram.
  - Its average values are **centered** around a **quiet** sound level.
  - The **faster** you read it, the more the data looks like a wave.
  - It is more likely to read both high **and** low values the longer a sound lasts.
- ## The EMA Calculation
  - This type of average puts more **significance**, or weight, on the last sample than previous samples.
  - The way sound waves go high and low at the same time means the average should stay consistent.
  - It doesn't require complicated math.
  - EMA uses a percentage to **weight** the new value more and the old values less.
  - The calculation looks like this:

```
new_average = (old_average * (1-WEIGHT)) + (new_value * WEIGHT)
```

Goals:

- Define the function `calc_avg(avg, new_val)`.

- Assign the variable `avg_sound`.
- `return avg_sound`.
- Assign `avg_sound` as the output of `calc_avg()`.
- Print `avg_sound`.

Tools Found: Variables, Print Function

Solution:

```
from codex import *
from time import sleep

# Constants for peripherals
LED_ON = 2**15        # Half-power duty cycle
LED_OFF = 0
TEMP_THRESHOLD = 2   # Degrees Celsius
WEIGHT = 0.02        # each new raw sound value has a 2% significance

# Set up the peripherals
led = exp.pwm_out(exp.PORT0, frequency=2)
temp_sensor = exp.analog_in(exp.PORT1)
sound_sensor = exp.analog_in(exp.PORT2)

def set_led(val):
    led.duty_cycle = val

def convert_temp_to_c(raw_temp):
    # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
    degrees_c = raw_temp * 0.004577 - 50
    return degrees_c

# calculates average of raw sound values using EMA
def calc_avg(avg, new_val):
    avg_sound = avg * (1 - WEIGHT) + new_val * WEIGHT
    return  avg_sound

def set_alarm():
    global temp_limit, avg_sound
    first_temp = convert_temp_to_c(temp_sensor.value)
    # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
    temp_limit = first_temp + TEMP_THRESHOLD
    # Initial sound reading
    avg_sound = sound_sensor.value

# Main Program
set_alarm()
while True:
    degree_c = convert_temp_to_c(temp_sensor.value)
    avg_sound = calc_avg(avg_sound, sound_sensor.value)
    print("Avg Sound: ", avg_sound)
    sleep(0.5)

    if degree_c > temp_limit:
        print("High Temperature Alert!")
        set_led(LED_ON)
        break
    else:
        set_led(LED_OFF)

# After the loop breaks, keep the LED flashing
while True:
    set_led(LED_ON)
```

Quiz 2 **- Check Your Understanding: Noise Detection**

Question 1: The sound sensor is NOT accurate enough to detect:

✓ a specific word

- ✗ an explosion
- ✗ two claps
- ✗ a loud bang

Question 2: The sound sensor will return a ____ value when the sound intensity increases?

- ✓ higher and lower
- ✗ higher
- ✗ lower
- ✗ average

Question 3: The temperature sensor will return a ____ value when the temperature increases?

- ✓ higher
- ✗ lower
- ✗ higher and lower
- ✗ average

Objective 8 - Put It All Together

You can now put the whole alerting system together!

First, modify your noise alert. Use this algorithm:

- Set a constant `LOUD_THRESHOLD` to indicate an explosion.
  - This will be the amount over or under the sound average that causes alarm.
- Calculate the **variation** of the last sound value from the average.
  - You have the sound average, so you should use it.
  - Subtract the average from the new raw sound value to get variation.
- Test for **loud** sounds using a single `if` statement.
  - sensor value is **HIGH** `if variation > LOUD_THRESHOLD` **or**
  - sensor value is **LOW** `if variation < -LOUD_THRESHOLD`
- Display a warning message and exit the loop when a loud noise is detected.

Second, delete unnecessary code from the `set_LED()` and temperature alert:

- An `else` is not needed as part of the `if` statement.
- Check the Hints for more details.

Check the 'Trek!

Follow the CodeTrek to combine the two alerts into one life-saving alert system.

Physical Interaction: *Try it!*

Run the code. Open the **Console Panel** and watch the output.

- Trigger the temperature alert.
- Run the code again.
- Trigger the noise alert.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  LED_ON = 2**15        # Half-power duty cycle
6  LED_OFF = 0
7  TEMP_THRESHOLD = 2    # Degrees Celsius
8  LOUD_THRESHOLD = 75   # Sound level variation for alert
```

Define `LOUD_THRESHOLD` as the amount above or below the sound average.

- Use `75` as the initial value, but you can adjust it if needed.

```
9  WEIGHT = 0.02        # each new raw sound value has a 2% significance
10
11 # Set up the peripherals
12 led = exp.pwm_out(exp.PORT0, frequency=2)
13 temp_sensor = exp.analog_in(exp.PORT1)
14 sound_sensor = exp.analog_in(exp.PORT2)
15
16 def set_led(val):
17     led.duty_cycle = val
18
19 def convert_temp_to_c(raw_temp):
20     # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
21     degrees_c = raw_temp * 0.004577 - 50
22     return degrees_c
23
24 # calculates average of raw sound values using EMA
25 def calc_avg(avg, new_val):
26     avg_sound = avg * (1 - WEIGHT) + new_val * WEIGHT
27     return  avg_sound
28
29 # Set starting values of global variables
30 def set_alarm():
31     global temp_limit, avg_sound
32     first_temp = convert_temp_to_c(temp_sensor.value)
33     # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
34     temp_limit = first_temp + TEMP_THRESHOLD
35     # Initial sound reading
36     avg_sound = sound_sensor.value
37
38 # Main Program
39 set_alarm()
40 while True:
41     degree_c = convert_temp_to_c(temp_sensor.value)
42     variation = sound_sensor.value - avg_sound
```

Calculate the variation in sound value.

- This step needs to happen at the beginning of each iteration.

```
43     print("Variation:", variation,"Temp: ", degree_c)
```

Modify the `print()` statement to display both `variation` and `degree_c`.

```
44     # Noise alert
45     if variation > LOUD_THRESHOLD or variation < -LOUD_THRESHOLD:
46         print("Alert - Explosion detected!")
47         set_led(LED_ON)
48         break
```

Add an `if` statement for the noise alert.

- Use **one** `if` statement with **two** conditions.
- Display an alert message, turn on the LED, and `break`.

```
49     # Temperature alert
50     if degree_c > temp_limit:
```

```
51          print("Alert - High Temperature")
52          set_led(LED_ON)
53          break
```

Delete the `else:` statement from the temperature alert.

```
54
55      # Re-calculate average sound after every read
56      avg_sound = calc_avg(avg_sound, sound_sensor.value)
57      # Wait one second between every sensor read
58      sleep(1)
```

Move the `avg_sound` update to **below** the `if` statements.

- Add a **one second** delay so the output is not overwhelming.

```
59
60  while True:
61      # Keep the LED blinking until reset (program quits)
62      set_led(LED_ON)
```

Hint:

- ## No Need for Else

  When the alarm is set, the LED will be off.

  - It only turns **ON** when the alarm is triggered.
  - When the alarm is triggered, the loop stops and the sensors are no longer read.
  - Then the crew must handle the emergency and reset the alarm.

  So neither `if` statement in the `while True:` loop **or** `set_led()` needs an `else`.

Goals:

- Calculate the **variation** and assign it to the variable `variation`.
- Use an `if` statement with the condition `variation > LOUD_THRESHOLD or variation < -LOUD_THRESHOLD`.
- Print `"Alert - Explosion Detected!"`.

Tools Found: Loops, Variables, bool, Print Function

Solution:

```
1   from codex import *
2   from time import sleep
3
4   # Constants for peripherals
5   LED_ON = 2**15          # Half-power duty cycle
6   LED_OFF = 0
7   TEMP_THRESHOLD = 2      # Degrees Celsius
8   LOUD_THRESHOLD = 75     # Sound level variation for alert
9   WEIGHT = 0.02           # each new raw sound has a 2% significance
10
11  # Set up the peripherals
12  led = exp.pwm_out(exp.PORT0, frequency=2)
13  temp_sensor = exp.analog_in(exp.PORT1)
14  sound_sensor = exp.analog_in(exp.PORT2)
15
16  def set_led(val):
17      led.duty_cycle = val
18
19  def convert_temp_to_c(raw_temp):
20      # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
```

```
21      degrees_c = raw_temp * 0.004577 - 50
22      return degrees_c
23
24  # calculates average of raw sound values using EMA
25  def calc_avg(avg, new_val):
26      avg_sound = avg * (1 - WEIGHT) + new_val * WEIGHT
27      return avg_sound
28
29  # Set starting values of global variables
30  def set_alarm():
31      global temp_limit, avg_sound
32      first_temp = convert_temp_to_c(temp_sensor.value)
33      # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
34      temp_limit = first_temp + TEMP_THRESHOLD
35      # Initial sound reading
36      avg_sound = sound_sensor.value
37
38  # Main Program
39  set_alarm()
40  while True:
41      degree_c = convert_temp_to_c(temp_sensor.value)
42      variation = sound_sensor.value - avg_sound
43      print("Variation: ", variation, "Temp: ", degree_c)
44
45      # Noise Alert
46      if variation > LOUD_THRESHOLD or variation < -LOUD_THRESHOLD:
47          print("Alert - Explosion Detected!")
48          set_led(LED_ON)
49          break
50      # Temperature Alert
51      if degree_c > temp_limit:
52          print("High Temperature Alert!")
53          set_led(LED_ON)
54          break
55
56      # Re-calculate average sound after every read
57      avg_sound = calc_avg(avg_sound, sound_sensor.value)
58      # Wait one second between every sensor read
59      sleep(1)
60
61  # After the loop breaks, keep the LED flashing until reset
62  while True:
63      set_led(LED_ON)
```

Objective 9 - Complete Alert System

Using the power of CodeX, you can add even more features to your alert system!

The CodeX has many built-in features that can enhance the alert system.

- Built-in speaker
- Four pixel LEDs
- Display screen
- And much more!

To really get the crew's attention, add a siren and turn on the pixel LEDs, in addition to the flashing red LED. Display the warning message on the screen.

- You will need to import the sound library and define a variable for the tone.

Check the 'Trek!

Follow the CodeTrek to add an alarm and extra features to the alert system.

Physical Interaction: *Try it!*

Run the code. Open the **Console Panel** and watch the output.

- Trigger the temperature alert.

- Trigger the noise alert.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3  from soundlib import *
```

Import the sound library

```
4
5  siren = soundmaker.get_tone("violin")
```

Define a variable for the tone.

- This example uses violin tone, but you can experiment with different tones.

```
6
7  # Constants for peripherals
8  LED_ON = 2**15        # Half-power duty cycle
9  LED_OFF = 0
10 TEMP_THRESHOLD = 2    # Degrees Celsius
11 LOUD_THRESHOLD = 75   # Sound level variation for alert
12 WEIGHT = 0.02         # each new raw sound has a 2% significance
13
14 # Set up the peripherals
15 led = exp.pwm_out(exp.PORT0, frequency=2)
16 temp_sensor = exp.analog_in(exp.PORT1)
17 sound_sensor = exp.analog_in(exp.PORT2)
18
19 def set_led(val):
20     led.duty_cycle = val
21
22 def convert_temp_to_c(raw_temp):
23     # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
24     degrees_c = raw_temp * 0.004577 - 50
25     return degrees_c
26
27 # calculates average of raw sound values using EMA
28 def calc_avg(avg, new_val):
29     avg_sound = avg * (1 - WEIGHT) + new_val * WEIGHT
30     return  avg_sound
31
32 # Siren warning sound for alarm, with pixel LEDs
33 def warning(pix_color):
```

Define the function `warning()` for the warning sound and colored pixels.

- It uses one parameter for the color of the pixel LEDs.

```
34     pixels.set([pix_color, pix_color, pix_color, pix_color])
```

Set all the pixel LEDS to `pix_color`.

```
35     siren.set_pitch(440)
36     siren.play()
37     siren.glide(880, 1.5)
38     sleep(1.5)
39     siren.glide(440, 1.5)
40     sleep(1.5)
41     siren.stop()
```

This is an example of code that sounds an alarm.

- You can experiment with different pitches, tones and delays.

```
42
43 # Set starting values of global variables
44 def set_alarm():
45     global temp_limit, avg_sound
46     first_temp = convert_temp_to_c(temp_sensor.value)
47     # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
48     temp_limit = first_temp + TEMP_THRESHOLD
49     # Initial sound reading
50     avg_sound = sound_sensor.value
51
52 # Main Program
53 set_alarm()
54
55 while True:
56     degree_c = convert_temp_to_c(temp_sensor.value)
57     variation = sound_sensor.value - avg_sound
58     print("Variation:", variation,"Temp:", degree_c)
59
60     # Noise alert
61     if variation > LOUD_THRESHOLD or variation < -LOUD_THRESHOLD:
62         display.print("Alert!", scale=5)
63         display.print("Explosion!", scale=3)
64         set_led(LED_ON)
65         pix_color = ORANGE
```

In the noise alert, modify the `print()` statement to display on the screen.

- Use `display.print()`.
- You can include a scale argument to make the text bigger.
- Select a color for the pixel LEDs.

```
66         break
67     # Temperature alert
68     if degree_c > temp_limit:
69         #TODO: display a print message for the alert
70         set_led(LED_ON)
71         #TODO: set pix_color
```

In the temperature alert, modify the `print()` statement to display on the screen.

- Use `display.print()`.
- You can include a scale argument to make the text bigger.
- Select a color for the pixel LEDs.

```
72         break
73
74     # Re-calculate average sound after every read
75     avg_sound = calc_avg(avg_sound, sound_sensor.value)
76     # Wait one second between every sensor read
77     sleep(1)
78
79 while True:
80     # Keep the LED blinking until reset (program quits)
81     set_led(LED_ON)
82     warning(pix_color)
```

Call the `warning()` function in the final infinite loop.

Hint:

- ## Coding Challenge

  Using the power of the CodeX with the peripherals, you can make a truly awesome alert system. Some extra features you can add to the system are:

  - Display a color or image on the screen to indicate the type of alarm.
  - Flash the pixel LEDS as well as the red LED.

- Use two different warning sounds, one for each alert.

Goals:

- 🔧Import the sound library `soundlib`.
- Define the 🔧function `warning(pix_color)`.
- Call `siren.play()`.
- Call `display.print()`.
- Call `warning()`.

Tools Found: import, Functions

Solution:

```
from codex import *
from time import sleep
from soundlib import *

siren = soundmaker.get_tone("violin")

# Constants for peripherals
LED_ON = 2**15        # Half-power duty cycle
LED_OFF = 0
TEMP_THRESHOLD = 2    # Degrees Celsius
LOUD_THRESHOLD = 75   # Sound level variation for alert
WEIGHT = 0.02         # each new raw sound has a 2% significance

# Set up the peripherals
led = exp.pwm_out(exp.PORT0, frequency=2)
temp_sensor = exp.analog_in(exp.PORT1)
sound_sensor = exp.analog_in(exp.PORT2)

def set_led(val):
    led.duty_cycle = val

def convert_temp_to_c(raw_temp):
    # deg C = ((raw temp / 65536) * 3000 mV  - 500 mV) / 10 deg per mV
    degrees_c = raw_temp * 0.004577 - 50
    return degrees_c

# calculates average of raw sound values using EMA
def calc_avg(avg, new_val):
    avg_sound = avg * (1 - WEIGHT) + new_val * WEIGHT
    return avg_sound

# Siren warning sound for alarm, with pixel LEDs
def warning(pix_color):
    pixels.set([pix_color, pix_color, pix_color, pix_color])
    siren.set_pitch(440)
    siren.play()
    siren.glide(880, 1.5)
    sleep(1.5)
    siren.glide(440, 1.5)
    sleep(1.5)
    siren.stop()

# Set starting values of global variables
def set_alarm():
    global temp_limit, avg_sound
    first_temp = convert_temp_to_c(temp_sensor.value)
    # Add TEMP_THRESHOLD (2 degrees) to initial temperature for alert
    temp_limit = first_temp + TEMP_THRESHOLD
    # Initial sound reading
    avg_sound = sound_sensor.value

# Main Program
set_alarm()
```

```
54 while True:
55     degree_c = convert_temp_to_c(temp_sensor.value)
56     variation = sound_sensor.value - avg_sound
57     print("Variation: ", variation, "Temp: ", degree_c)
58 
59     # Noise Alert
60     if variation > LOUD_THRESHOLD or variation < -LOUD_THRESHOLD:
61         display.print("Alert!", scale=5)
62         display.print("Explosion!", scale=3)
63         set_led(LED_ON)
64         pix_color = ORANGE
65         break
66     # Temperature Alert
67     if degree_c > temp_limit:
68         display.print("Alert!", scale=5)
69         display.print("High Temp!", scale=3)
70         set_led(LED_ON)
71         pix_color = RED
72         break
73 
74     # Re-calculate average sound after every read
75     avg_sound = calc_avg(avg_sound, sound_sensor.value)
76     # Wait one second between every sensor read
77     sleep(1)
78 
79 # After the loop breaks, keep the LED flashing until reset
80 while True:
81     set_led(LED_ON)
82     warning(pix_color)
```

Mission 5 Complete

You've completed project Alert System!

The shuttle is now a much safer environment for the crew. The crew will rest easier knowing that their safety critical systems are being monitored.

Mission 6 - Life Support!

This project shows you how to operate a 360 continuous servo to circulate life-giving air through the ship!

Mission Briefing:

The air we breathe is one of the most fundamental elements of human life.

- The spacecraft life support systems must be operational to guarantee mission success!

The circulation of air is critical to survival in space.

- It mixes oxygen with carbon dioxide to guarantee breathable air.
- It collects condensation on the ship's hull and recycles it.
- It mixes warm and cold air for the perfect environment for both crew and electronics.

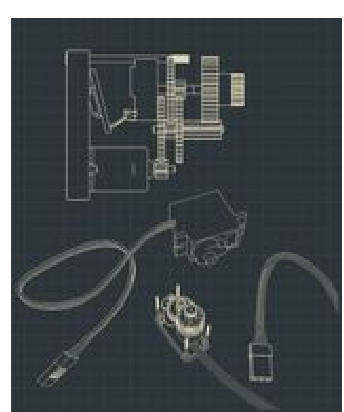
Project: Life Support will guide you through rotating fans to guarantee proper air circulation on the ship.

Project Goals:

- Learn about 360 Continuous Rotation Servos
- Understand the capabilities of a 360 Servo
- Add a power switch to the life support system
- Explore the concept of a finite-state machine (FSM)
- Circulate air to support the crew on their journey through space!!

Objective 1 - Introduction to Servos

You will need fans to operate the ship's life support system and circulate air.



When you think about a servo motor you probably assume that it is nothing more than a DC electrically operated motor.

This is sort of true.

A servo is more than just a motor. It contains:

- A DC motor
- A controller circuit
- An internal feedback mechanism
- A gearbox

Small hobby servos generally come it two types:

- The **360 Continuous Rotation Servo** which can rotate continuously backward and forward.
- The **180 Positional Servo** which can move to a specified position and hold in place.

You will need fans to operate the ship's life support system and circulate air.

- The 360 servo can drive a fan to circulate air through the spacecraft.
- Check the Hints for more details about servos.

Mechanical Connection to Servos

The rotating drive-shaft of a servo has ridges called *splines* that allow to you securely attach *wheels*, *arms*, or *servo-horns* to move things.

- There's a servo-horn in your kit!
- Try attaching it to your servo so you can better see the motion.

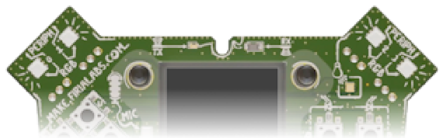
Connect Peripheral

Connect the **360 Servo** to **PORT0** on your CodeX.

- Disconnect all other peripherals from the CodeX.
- Attach **one** *servo-horn* to the servo.

NOTE: Your servo will have orange, red, and brown wires.

- The **ORANGE** wire is inserted into the **S**
- The **BROWN** wire is inserted into the **G**



How do I make the servo go?

Start by turning the servo clockwise. Servo motors require an **analog control signal** to operate.

- You can send an analog control signal using PWM!
- You will learn how the control signal works later in the project.
- Start by getting it moving!!

Create a New File!

Use the **File** → **New File** menu to create a new file called ***LifeSupport***.

Check the 'Trek!

Follow the CodeTrek to set up the servo and get it moving.

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep
```

Import the libraries you will need.

```
3
4  # Constants for peripherals
5  PERIOD = 20
```

Define a constant for the frequency.

- `PERIOD` is used by the servo as part of PWM.
- Check the Hints for more information about `PERIOD`.

```
6
7  # Set up peripherals
```

```
8  fan = exp.pwm_out(exp.PORT0, frequency=PERIOD)  # 360 servo
```

Set up the 360 servo on PORT0.

- Use PWM.
- The servo is an output peripheral.

```
9
10 # Main program
11 while True:
12     fan.duty_cycle = 1000
```

Use a `while True:` loop to make the fan turn continuously.

- Right now, just assign a value to the duty_cycle.
- You can try different values and see what happens.

Hints:

- ## DC Motors

  A DC motor can come in various configurations, but in the end it is just a motor.

  - When power is applied the motor turns.

  A 🔧servo is a **DC motor** with more components.

- ## Directionally Challenged

  360 Servos have no sense of **position**, but they have the ability to rotate forward or backward continuously.

  Continous Rotation Servos and DC Motors could have many REAL uses on a space mission:

  - Motors for rover wheels
  - Fans for life support
  - Hydraulic pump operation
  - Spin antennas / dishes
  - Reel in mechanical components

- ## Servo Control Signal

  Nearly all servos operate with a **50 Hertz (Hz)** control signal, or PWM.

  - 50 Hz = 50 pulses per second
  - 50 Hz became a standard long ago due to the simplicity of the hardware design.
  - And, well, it just stuck...

  So how long does it take for one pulse?

  - PERIOD = 1 second / 50 Hz
  - PERIOD = 0.02 seconds
  - or a 20 millisecond analog PERIOD.

- ## Servo Accessories

  The peripherals kit comes with a bag of parts for the servo. You will just need one *servo-horn* for this mission.

Goals:

- Create a new file named `LifeSupport`.
- Declare the constant `PERIOD`.
- Set up the **360 Servo** by assigning the variable `fan` as the output of `exp.pwm_out(exp.PORT0, frequency=FREQ)`.
- Assign `fan.duty_cylce` as `1000`.

Tools Found: Servos, PWM, Constants, Variables

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for the peripherals
5  PERIOD = 20
6
7  # Set up the peripherals
8  fan = exp.pwm_out(exp.PORT0, frequency=PERIOD)
9
10 # Main program
11 while True:
12     fan.duty_cycle = 1000
```

Quiz 1 **- Check Your Understanding: Servos**

Question 1: Which type of device moves to a specified position and holds its place?

✓ 180 Servo

✗ 360 Servo

❌ Water Pump

❌ Potentiometer

Question 2: Which type of devices can rotate continuously forward and backward?

✅ 360 Servo

❌ 180 Servo

❌ Water Pump

❌ Potentiometer

Question 3: Which is **NOT** part of a servo motor?

✅ Speaker

❌ Controller Circuit

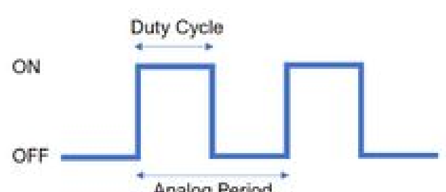
❌ DC Motor

❌ Gearbox

Objective 2 - Analog Control Signal

Time to Learn about Servo Analog Control!

How did your servo driving code work?



PWM uses a frequency, or **analog period**.

- In the code, you set the analog period to 20 milliseconds/Hz.
- Check the 💡 Hints for more details.

The **duty_cycle** sets the time the signal is on. The CodeX uses 16 bits (0-65535).

You want the servo to be

- **ON** for 1 millisecond
- **OFF** for 19 milliseconds so the calculation is:

CYCLE = 2^{16} / PERIOD

The **360 Continous Rotation Servo** can rotate continuously backward and forward.

- You can control the servo by adjusting the percentage of the CYCLE.

Percent of CYCLE	Speed of Motor	Rotation Direction
20	100%	Clockwise
40	50%	Clockwise
60	0%	Stopped
80	50%	Counterclockwise
100	100%	Counterclockwise

Create a function to turn on the servo by calculating the duty_cycle for the speed and rotation direction.

Check the 'Trek!

Follow the CodeTrek to:

- Define a function that calculates the duty_cycle.
- Turn the fan forward, or clockwise, and then stop.
- And then turn the fan backwards, or counterclockwise, and then stop.

Run It!

The fan should rotate forward, stop, rotate backward, and stop.

- You can change the percentages to see the change in speed.
- The servo will just stop moving if the limit is exceeded.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20            # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
```

Define a constant for `CYCLE`.

- The constant will simplify the calculation for the duty_cycle.
- The `//` division operator returns an integer instead of a float (or decimal).

```
7
8  # Set up peripherals
9  fan = exp.pwm_out(exp.PORT0, frequency=PERIOD)  #360 Servo
10
11 # Set the servo by calculating the duty_cycle
12 def set_servo(percent):
13     return CYCLE * percent // 100
```

Define `set_servo()` to return the duty_cycle value for the direction and speed.

- It takes one parameter `percent`.
- Calculate the percent of `CYCLE`.
- The `//` division operator returns an integer instead of a float (or decimal).

```
14
15 while True:
16     # Turn fan forwards at 50%
17     fan.duty_cycle = set_servo(40)
18     sleep(3)
```

Call `set_servo()` to turn the fan forwards.

- For the argument, use a percent in the range for clockwise rotation.
- Include a short delay to watch the rotation.

```
19     # Stop the fan
20     fan.duty_cycle = set_servo(60)
21     sleep(3)
```

Call `set_servo()` to stop the fan.

- For the argument, use a percent in the range for stopping.
- Include a short delay.

```
22     # Turn the fan backwards at 50%
23     fan.duty_cycle = set_servo(80)
24     sleep(3)
25     # Stop the fan
26     fan.duty_cycle = set_servo(60)
27     sleep(3)
```

Call `set_servo()` to turn the fan backwards.

- For the argument, use a percent in the range for counterclockwise rotation.
- Include a short delay to watch the rotation.
- Then call `set_servo()` again to stop the fan.
- Include a short delay.

Hints:

- ## Servo Control Signal

  Nearly all servos operate with a **50 Hertz (Hz)** control signal, or PWM.

  - 50 Hz = 50 pulses per second

  So how long does it take for one pulse?

  - PERIOD = 1 second / 50 Hz
  - PERIOD = 0.02 seconds, or a 20 milliseconds / Hz

- ## Duty Cycles

  The Servo on the CodeX takes a value between 0 and $2^{16} - 1$

  In each analog period, you want the signal to be:

  - **ON** for 1 millisecond
  - **OFF** for 19 milliseconds

  One CYCLE = $2^{16}/PERIOD$

  - A percentage of one CYCLE will produce varying speeds in both directions.

- ## Max and Min

  The servo has a minimum and maximum `duty_cycle` limit.

  The servo will just stop moving if the limit is extended in either direction.

Goals:

- Define the constant `CYCLE`.
- Define the function `set_servo(percent)`.
- Assign `fan.duty_cycle` *at least* three times.
- Call `set_servo()` *at least* three times.

Tools Found: Constants, Functions

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for the peripherals
5  PERIOD = 20              # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7
8  # Set up the peripherals
9  fan = exp.pwm_out(exp.PORT0, frequency=PERIOD)  # 360 servo
10
11 # Set the servo by calculating the duty_cycle
12 def set_servo(percent):
13     return CYCLE * percent // 100
14
15 # Main program
16 while True:
```

```
17      # Turn fan forwards at 50%
18      fan.duty_cycle = set_servo(40)
19      sleep(3)
20      # Stop the fan
21      fan.duty_cycle = set_servo(60)
22      sleep(3)
23      # Turn the fan backwards at 50%
24      fan.duty_cycle = set_servo(80)
25      sleep(3)
26      # Stop the fan
27      fan.duty_cycle = set_servo(60)
28      sleep(3)
```

Objective 3 - Down for Maintenance!

The crew needs the ability to shut down the fan periodically to perform maintenance.

The circulation system should have two different operating modes:

- Active
- Maintenance

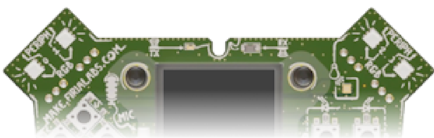
You can use a switch to disable the circulation system temporarily.

The algorithm to transition between the modes looks like this:

- `if` the switch is **ON** - Run the fan (Active Mode)
- `if` the switch is **OFF** - Stop the fan (Maintenance Mode)

Connect Peripheral

Connect the **switch** to **PORT2** on your CodeX.



Check the 'Trek!

Follow the CodeTrek to add the switch as a control.

Physical Interaction: *Try it!*

- Press the switch IN to the ON position.
- Press the switch OUT to the OFF position.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
```

```
5  PERIOD = 20              # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  POWER_ON = False         # Switch
8  POWER_OFF = True         # Switch
```

Define two **constants** for the **switch**.

- `True` is returned when the switch is **out**.
- `False` is returned when the switch is **in**.

```
9
10 # Set up peripherals
11 fan = exp.make_pwm(exp.PORT0, frequency=PERIOD) # 360 servo
12 switch = exp.digital_in(exp.PORT2)
```

Set up the **switch** on **PORT2**.

- It is a digital input peripheral.

```
13
14 # Set the servo by calculating the duty_cycle
15 def set_servo(percent):
16     return CYCLE * percent // 100
17
18 while True:
19     # Turn fan forwards at 50%
20     if switch.value == POWER_ON:
21         fan.duty_cycle = set_servo(40)
```

If the switch is **ON**, turn the fan forward.

- Call `set_servo()` to set the duty_cycle.
- For the argument, use a percent in the range of clockwise rotation.

```
22     # Stop the fan
23     if switch.value == POWER_OFF:
24         fan.duty_cycle = set_servo(60)
```

If the switch is **OFF**, turn the fan off.

- Call `set_servo()` to set the duty_cycle.
- For the argument, use a percent in the range of stopping.
- Delete all other code from the `while True:` loop.

Goals:

- Define constants `POWER_ON` and `POWER_OFF`.
- Set up the **switch** by assigning the variable `switch` as the output of `exp.digital_in(exp.PORT2)`.
- Use an `if` statement with the condition `switch.value == POWER_ON`.
- Use an `if` statement with the condition `switch.value == POWER_OFF`.

Tools Found: Constants, Variables, bool

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for the peripherals
5  PERIOD = 20              # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  POWER_ON = False         # Switch
8  POWER_OFF = True         # Switch
```

```
9
10  # Set up the peripherals
11  fan = exp.pwm_out(exp.PORT0, frequency=PERIOD)  # 360 servo
12  switch = exp.digital_in(exp.PORT2)
13
14  # Set the servo by calculating the duty_cycle
15  def set_servo(percent):
16      return CYCLE * percent // 100
17
18  # Main program
19  while True:
20      # Turn fan forwards at 50%
21      if switch.value == POWER_ON:
22          fan.duty_cycle = set_servo(40)
23      # Stop the fan
24      if switch.value == POWER_OFF:
25          fan.duty_cycle = set_servo(60)
```

Objective 4 - Software States!

Time to add more definition to the system's modes.

You already set up two different operating modes:

- Active
- Maintenance

But, what if you wanted to add a third mode... or a fourth... and there were many conditions involved?

- You could use a concept called the Finite-State Machine.

Concept

Finite-State Machine

A Finite-State Machine, sometimes called simply a State Machine, is a key concept in software and hardware systems.

It is the idea that a program can only be in **one of a known set** of states or "phases" at any given time.

- Keeping track of states helps you as a programmer understand and manage your code.
- The software can transition from one state to another when certain conditions are met.
- Check the Hints to see an example of a Finite-State Machine.

Time to Apply a Finite-State Machine to the Circulation System.

Check the 'Trek!

CodeTrek:

```
1   from codex import *
2   from time import sleep
3
4   # Constants for the peripherals
5   PERIOD = 20                 # Milliseconds / Hz
6   CYCLE = 2**16 // PERIOD
7   POWER_ON = False            # Switch
8   POWER_OFF = True            # Switch
9   FORWARD = 40                # Percent for servo 50% clockwise
10  STOP = 60                   # Percent for servo stopping
```

Define constants for FORWARD and STOP.

- The constants are optional, but they are good practice.
- Assign each constant the percent for each rotation.
- No more *magic numbers*!

```
11
12  # Set up peripherals
13  fan = exp.make_pwm(exp.PORT0, frequency=PERIOD) # 360 servo
14  switch = digitalio.DigitalInOut(exp.PORT2)
15
16  # Set the servo by calculating the duty_cycle
17  def set_servo(percent):
18      return CYCLE * percent // 100
19
20  # Main program
21  # Define variable for indicating the current state and
22  # set the initial duty_cycle to forward
23  state = "active"
24  fan.duty_cycle = set_servo(FORWARD)
```

Define the `state` variable and initialize it to `"active"`.

- Set the initial `fan.duty_cycle` to a forward speed.

```
25
26  while True:
27      # Current state is "off", so transition to "on"
28      if state == "maintenance":
29          if switch.value == POWER_ON:
30              state = "active"
31              fan.duty_cycle = set_servo(FORWARD)
```

Modify the `while True:` loop to use `state` in the `if` statement.

- Compare `state` to `"maintenance"`.
- If `True`, read the switch.
- When the switch is turned **ON**, transition to the next state by updating `state`.
- Set `fan.duty_cycle` to a forward speed.

```
32      # Current state is "on", so transition to "off"
33      elif state == "active":
34          if switch.value == POWER_OFF:
35              state = "maintenance"
36              fan.duty_cycle = set_servo(STOP)
```

Add an `elif` branch to the `if` statement to check the second state.

- Compare `state` to `"active"`.
- If `True`, read the switch.
- When the switch is turned **OFF**, transition to the next state by updating `state`.
- Set `fan.duty_cycle` to stop.

Hint:

- ## Finite-State Machine Example: Traffic Light

  Most traffic lights are three colors:

  - Red
  - Yellow
  - Green

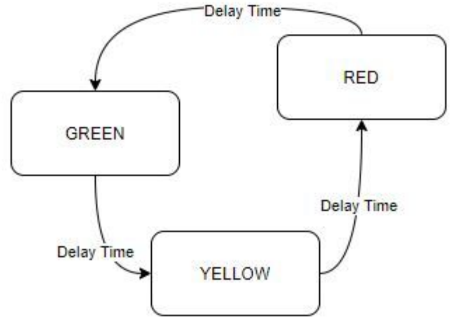
  Traffic lights, in the United States of America, generally only have three states:

  - GREEN = Traffic can go
  - YELLOW = Caution the light will soon be red
  - RED = Traffic must stop

  The 🔧transitions are simple and all transitions occur after a time delay:

- GREEN only transitions to YELLOW
- YELLOW only transitions to RED
- RED only transitions to GREEN

Here is a visual representation of a 🔧state machine for a traffic light in the USA:



Goals:

- Define the 🔧constants `FORWARD` and `STOP`.
- Use an `if` statment with the 🔧condition `state == "maintenance"`.
- Use an `elif` statment with the 🔧condition `state == "active"`.
- Assign `fan.duty_cycle` as `set_servo(FORWARD)` *at least* twice.

Tools Found: bool, State, Loops, Constants

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for the peripherals
5  PERIOD = 20                # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  POWER_ON = False           # Switch
8  POWER_OFF = True           # Switch
9  FORWARD = 40               # Percent for servo 50% clockwise
10 STOP = 60                  # Percent for servo stopping
11
12 # Set up the peripherals
13 fan = exp.pwm_out(exp.PORT0, frequency=PERIOD) #360 servo
14 switch = exp.digital_in(exp.PORT2)
15
16 # Set the servo by calculating the duty_cycle
17 def set_servo(percent):
18     return CYCLE * percent // 100
19
20 # Main program
21 # Define variable for indicating the current state and
22 # set the initial duty_cycle to forward
23 state = "active"
24 fan.duty_cycle = set_servo(FORWARD)
25
26 while True:
27     # Current state is "off", so transition to "on"
28     if state == "maintenance":
29         if switch.value == POWER_ON:
30             state = "active"
31             fan.duty_cycle = set_servo(FORWARD)
32     # Current state is "on", so transition to "off"
33     elif state == "active":
34         if switch.value == POWER_OFF:
```

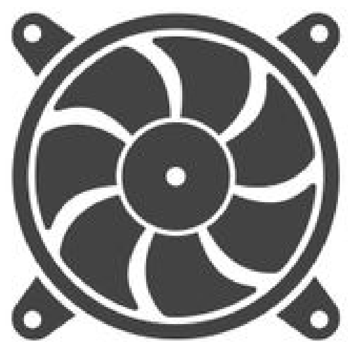
```
35              state = "maintenance"
36              fan.duty_cycle = set_servo(STOP)
```

Mission 6 Complete

You've completed project Life Support!

You got a chance to work with your first servo. The crew is thankful for the perfectly climate controlled cabin. They can now survive the long trip through space!

Great work!!



Mission 7 **- Solar Tracking!**

This project shows you how to operate a 180 positional servo to track the sun with solar panels!

Mission Briefing:

The space ship needs to harvest power from the sun.

- The more efficient the solar panels can be the better!

Rotating the panels to track the sun can improve power generation immensely.

Project: Solar Tracking will guide you through turning solar panels to track the sun.

Project Goals:

- Learn about 180 Positional Servos
- Understand the capabilities of a 180 Servo
- Add a light sensor to detect sun intensity
- Rotate the ship's solar panels to the sun's general location!!

Objective 1 **- 180 Servo**

The ship's solar panels can be rotated to track the sun.

- For this project, a sense of position matters.
- Also, you want to hold the solar panels' position for a while and not have it move continuously.
- You will need the ***180 servo*** for this task.
- 180 servos have a sense of **position** but cannot rotate a full 360 degrees.
- Check the Hints for more information about the **180 servo**.

Now attach the 180 servo!

- The 180 servo will rotate the solar panels to track the sun.

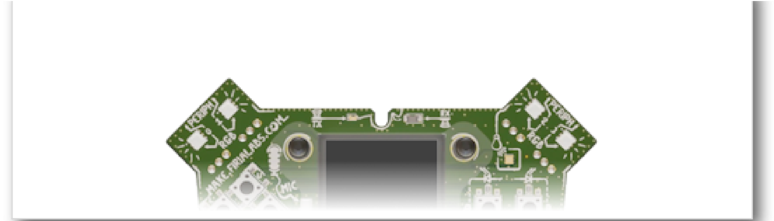
Connect Peripheral

Connect the **180 servo** to **PORT0** on your CodeX.

- Disconnect all other peripherals on the CodeX.
- Attach **one** *servo-horn* to the servo.

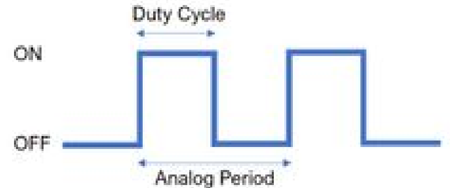
NOTE: Your servo will have orange, red, and brown wires.

- The **ORANGE** wire is inserted into the **S**
- The **BROWN** wire is inserted into the **G**



What are the duty cycle settings for the 180 servo?

The duty cycle for a **180 servo** determines which position it rotates to.



Percent of CYCLE	Angle of Rotation	Direction of Rotation
25	90 Degrees	Clockwise
50	45 Degrees	Clockwise
75	0 Degrees	Centered
100	45 Degrees	Counterclockwise
125	90 Degrees	Counterclockwise

Add a function to set the **duty cycle** for your servo.

Check the 'Trek!

Follow the CodeTrek to:

- Set up the **180** servo on **PORT0**.
- Define a function to calculate the duty_cycle.
- Turn the panels 45 degrees in each direction.

Run It!

The servo panels should move to two different positions:

- One rotating forward,
- One rotating backward.

CodeTrek:

```
1  from codex import *
2  from time import sleep
```

Import the libraries.

- You do this every mission!

```
3
4  # Constants for peripherals
5  PERIOD = 20               # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7
8  # Set up peripherals
9  panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
```

Define constants and set up the servo.

- Constants `PERIOD` and `CYCLE` are used by the servo.
- Set up the servo to use PWM on PORT0.

```
10
11 # Set the servo by calculating the duty_cycle
12 def set_servo(percent):
13     return CYCLE * percent // 100
```

Define the function `set_servo()` to calculate the duty_cycle.

- Use one parameter `percent`.
- This is the same function used in **Mission Life Support**.

```
14
15 while True:
16     # Rotate 45 degrees forward and then stop
17     panels.duty_cycle = set_servo(50)
18     sleep(2)
19     panels.duty_cycle = set_servo(75)
20     sleep(2)
```

Set the panel to 45 degrees forward for 2 seconds.

Then return to center for 2 seconds.

```
21     # Rotate 45 degrees backward and then stop
22     panels.duty_cycle = set_servo(100)
23     sleep(2)
24     panels.duty_cycle = set_servo(75)
25     sleep(2)
```

Set the panel to 45 degrees backward for 2 seconds.

Then return to center for 2 seconds.

Hints:

- ## Positional Servos

  180 servos can have many REAL uses on a space mission:
- Rotating solar panels
- Controlling robotic arms
- Moving latches or locks

- Positioning devices or antennas

- ## Duty Cycle Chart

  The chart for the **180 servo** is similar to the chart for the **360 servo**.
- Both use a percent of the CYCLE.
- Both can rotate forward and backward.
- The **180 servo** uses an angle instead of speed.
- The **180 servo** will **hold** its position instead of constantly spinning.

- If you push the servo horn, the **180 servo** will self-correct to the angle.

- ## Starting Code

  The program for this mission will start very much like the last mission.
- Import the libraries.
- Define constants for the peripherals.
- Set up the servo on PORT0 as a PWM.
- Define a function to calculate the duty cycle of the servo.

How much of this code can you do on your own?

Goals:

- Create a new file named `SolarTracking`.
- Set up the **180 servo** by assigning the 🔧variable `panels` as the output of `exp.pwm_out(exp.PORT0, frequency=FREQ)`.
- Define the 🔧function `set_servo(percent)`.
- Call `set_servo()` at least four times.

Tools Found: Servos, Variables, Functions

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20              # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7
8  # Set up peripherals
9  panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
10
11 # Set the servo by calculating the duty_cycle
12 def set_servo(percent):
13     return CYCLE * percent // 100
14
15 while True:
16     # Rotate 45 degrees forward and then stop
17     panels.duty_cycle = set_servo(50)
18     sleep(2)
19     panels.duty_cycle = set_servo(75)
20     sleep(2)
21     # Rotate 45 degrees backward and then stop
22     panels.duty_cycle = set_servo(100)
23     sleep(2)
24     panels.duty_cycle = set_servo(75)
25     sleep(2)
```

Objective 2 - Stay in Position!

The 180 servo will always strive to stay in position!

There is no **OFF** position for the **180** servo like there is for a 360 servo.

- The 180 servo is always working to stay in position.
- If you push it either direction it will always come back to its set position.

On the other hand...

It is normal for the **180 servo** to continuously make small movements and noise even after it gets in position.

- The 180 servo does not turn **OFF** while sending a signal.
- The only way to guarantee that the **180 servo** will stop moving is to stop the **PWM** signal completely.
- You will need to break out of the loop to stop the servo.
- This will also cause the **180 servo** to stop striving for position!

Check the 'Trek!

Follow the CodeTrek to stop the **180 servo** when the code is finished executing.

Physical Interaction

Run the code. While the servo is moving, push on the servo and try to move it to a different position.

- Press BTN_A to break the loop.
- The servo should stop.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20              # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  FORWARD = 50             # Percent for 45 degree angle clockwise
8  CENTER = 75              # Percent for 0 degree angle (centered)
9  BACKWARD = 100           # Percent for 45 degree angle counterclockwise
#@1
10
11 # Set up peripherals
12 panels = exp.make_pwm(exp.PORT0, frequency=PERIOD) # 180 servo
13
14 # Set the servo by calculating the duty_cycle
15 def set_servo(percent):
16     return CYCLE * percent // 100
17
18 while True:
19     # Rotate 45 degrees forward and then stop
20     panels.duty_cycle = set_servo(FORWARD)
21     sleep(2)
22     panels.duty_cycle = set_servo(CENTER)
23     sleep(2)
24     # Rotate 45 degrees backward and then stop
25     panels.duty_cycle = set_servo(BACKWARD)
26     sleep(2)
27     panels.duty_cycle = set_servo(CENTER)
```

Use the constants in the function calls to rotate the panels.

- No *magic numbers*!

```
28     sleep(2)
29
30     if buttons.was_pressed(BTN_A):
31       break
```

Use a button press to break out of the loop.

```
32
33 # Stop the servo
34 panels.duty_cycle = 0
```

Stop the servo by setting the duty_cycle to `0`.

Hints:

- ## 180 Servo

  A positional servo is great for a system like a flap on an airplane!

  - When the pilot moves the flap to a certain position he or she expects it to stay there.
  - They do not want the wind rushing onto the flap to move it around.

- ## Still ON!?

  The peripherals will stay on after the program stops running.

  - One way to solve this problem is to break out of the loop and then turn off the peripherals.
  - Use a button press to break the loop.

Goals:

- Define the constants `FORWARD`, `CENTER`, and `BACKWARD`.

- Call `set_servo(FORWARD)`, `set_servo(CENTER)`, and `set_servo(BACKWARD)`.
- Use an `if` statement with the 🔧condition `buttons.was_pressed(BTN_A)`.
- Assign `panels.duty_cycle` as `0`.

Tools Found: Servos, Constants, bool

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20                # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  FORWARD = 50               # Percent for 45 degree angle clockwise
8  CENTER = 75                # Percent for 0 degree angle (centered)
9  BACKWARD = 100             # Percent for 45 degree angle counterclockwise
10
11 # Set up peripherals
12 panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
13
14 # Set the servo by calculating the duty_cycle
15 def set_servo(percent):
16     return CYCLE * percent // 100
17
18 while True:
19     # Rotate 45 degrees forward and then stop
20     panels.duty_cycle = set_servo(FORWARD)
21     sleep(2)
22     panels.duty_cycle = set_servo(CENTER)
23     sleep(2)
24     # Rotate 45 degrees backward and then stop
25     panels.duty_cycle = set_servo(BACKWARD)
26     sleep(2)
27     panels.duty_cycle = set_servo(CENTER)
28     sleep(2)
29
30     if buttons.was_pressed(BTN_A):
31         break
32
33 # Stop the servo
34 panels.duty_cycle = 0
```

Quiz 1 - Check Your Understanding: 180 Servos

Question 1: A **180 servo** is also known as:

✓ A positional servo

✗ A continuous rotation servo

✗ A potentiometer

✗ A DC motor

Question 2: When will a positional servo stop moving to the correct position?

✓ When you stop sending a PWM signal.

✗ Once it gets in position.

✗ After a delay.

✗ When the next command is executed.

Question 3: What percent is used to **stop** the 180 servo?

`0`

`75`

`100`

`50`

Objective 3 **- Light Sensor**

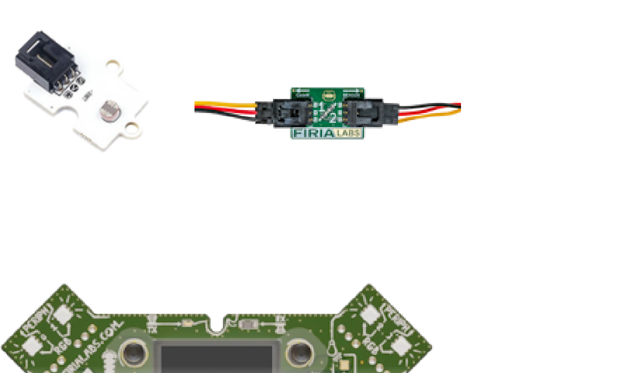
Now that you can rotate the solar panels to a specific location, you need to detect the sun to start tracking it.

Use the Light Sensor of course!

- The light sensor is an **analog input** that provides the **intensity** of ambient light.
- Like other analog input devices, the **light sensor** will use the resistor voltage divider to limit the ADC input.

Connect Peripheral

Connect the **divider** to **PORT1** and the **light sensor** to the **divider** on your CodeX.



Concept

The light sensor is built with a photoresistor.

- A photoresistor creates less **resistance** when light hits it.
- Less resistance allows more current to pass through.
- Your device reads a higher analog value in its ADC as more current flows.

MORE LIGHT = HIGHER VALUE

Time to output some values to see how the light sensor works!

Use Python's built-in `print()` function for printing to the **console panel**.

- You'll need to open the **Console Panel** in *CodeSpace* to use this function.
  - Use the ☰ button.
  - The Console Panel button is below the *toolbox button*.
- The **Console Panel** command line is called the **REPL**.
  - *"Read Evaluate Print Loop"*

Check the 'Trek!

Follow the CodeTrek to set up the light sensor and read its values.

Physical Interaction: *Try it*

Run the code. Open the **console panel** and watch the light sensor values as they are displayed.

- Shine a light on the sensor and get a high reading.
- Get the normal reading with the room lighting.
- Cover the sensor with your hands so it is completely dark and get the low reading.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20                # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  FORWARD = 50               # Percent for 45 degree angle clockwise
8  CENTER = 75                # Percent for 0 degree angle (centered)
9  BACKWARD = 100             # Percent for 45 degree angle counterclockwise
10
11 # Set up peripherals
12 panels = exp.make_pwm(exp.PORT0, frequency=PERIOD) # 180 servo
13 light_sensor = exp.analog_in(exp.PORT1)
```

Set up the light sensor on PORT1

- It is an analog input peripheral.

```
14
15 # Set the servo by calculating the duty_cycle
16 def set_servo(percent):
17     return CYCLE * percent // 100
18
19 # Use this loop for reading the light sensor values
20 while True:
21     print(light_sensor.value)
22     sleep(1)
```

Use a `while True:` loop to read the values of the light sensor.

- `print()` to the **console panel**.
- This loop is for testing the peripheral and reading values.

```
23
24 while True:
25     # Rotate 45 degrees forward and then stop
26     panels.duty_cycle = set_servo(FORWARD)
27     sleep(2)
28     panels.duty_cycle = set_servo(CENTER)
29     sleep(2)
30     # Rotate 45 degrees backward and then stop
31     panels.duty_cycle = set_servo(BACKWARD)
32     sleep(2)
33     panels.duty_cycle = set_servo(CENTER)
34     sleep(2)
35
36     if buttons.was_pressed(BTN_A):
37         break
38
39 # Stop the servo
40 panels.duty_cycle = 0
```

Hint:

- ## Light Sensor

  Ambient light sensors are used in all kinds of real-world applications:

  - Night lights
  - Solar street lamps
  - Car interior lights
  - Real-world solar tracking devices

Goals:

- Set up the **light sensor** by assigning the 🔧variable `light_sensor` as the output of `exp.analog_in(exp.PORT1)`.
- In a `while True:` 🔧loop:
- 🔧Print `light_sensor.value`
- `sleep()` for one second.

Tools Found: Analog to Digital Conversion, Print Function, REPL, Variables, Loops

Solution:

```
from codex import *
from time import sleep

# Constants for peripherals
PERIOD = 20               # Milliseconds / Hz
CYCLE = 2**16 // PERIOD
FORWARD = 50              # Percent for 45 degree angle clockwise
CENTER = 75               # Percent for 0 degree angle (centered)
BACKWARD = 100            # Percent for 45 degree angle counterclockwise

# Set up peripherals
panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
light_sensor = exp.analog_in(exp.PORT1)

# Set the servo by calculating the duty_cycle
def set_servo(percent):
    return CYCLE * percent // 100

# Use this loop for reading the light sensor values
while True:
    print(light_sensor.value)
    sleep(1)

while True:
    # Rotate 45 degrees forward and then stop
    panels.duty_cycle = set_servo(FORWARD)
    sleep(2)
    panels.duty_cycle = set_servo(CENTER)
    sleep(2)
    # Rotate 45 degrees backward and then stop
    panels.duty_cycle = set_servo(BACKWARD)
    sleep(2)
    panels.duty_cycle = set_servo(CENTER)
    sleep(2)

    if buttons.was_pressed(BTN_A):
        break

# Stop the servo
panels.duty_cycle = 0
```

Objective 4 - Analyzing Solar State

Time to match light intensity with the solar panel position.

Check the 'Trek!

Run It!

Run the code with

```
state = 'morning'
```

- The panels should hold the 45 degree forward position.
- Change the value of `state` to `'afternoon'` and run the code.
- Change the value of `state` to `'night'` and run the code.
- Do the panels hold the correct positions?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20                 # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  FORWARD = 50                # Percent for 45 degree angle clockwise
8  CENTER = 75                 # Percent for 0 degree angle (centered)
9  BACKWARD = 100              # Percent for 45 degree angle counterclockwise
10
11 # Set up peripherals
12 panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
13 light_sensor = exp.analog_in(exp.PORT1)
14
15 # Set the servo by calculating the duty_cycle
16 def set_servo(percent):
17     return CYCLE * percent // 100
18
19 # Main program
20 panels.duty_cycle = set_servo(FORWARD)
21 state = 'morning'
```

Define `state` and initialize it to `'morning'`.

- Set the initial panel position to `FORWARD`.
- This will begin the main program.

```
22
23 while True:
```

You can delete the `while True:` loop that read the light sensor values.

- It was used for testing.

```
24     if state == 'morning':
25         panels.duty_cycle = set_servo(FORWARD)
26     elif state == 'afternoon':
27         panels.duty_cycle = set_servo(CENTER)
28     elif state == 'night':
29         panels.duty_cycle = set_servo(BACKWARD)
```

Modify the `while True:` loop to use `state` to control the solar panels.

- Use an if statement with three branches, one for each **state**.
- Delete all other code from the loop.
- But keep the button press! You need to break the loop to stop the peripherals.

```
30
```

```
31      if buttons.was_pressed(BTN_A):
32          break
33
34  # Stop the servo
35  panels.duty_cycle = 0
```

Hints:

- ## Tracking Sunlight

  In the real-world you would probably need:

  - A complicated algorithm to monitor the ship's orientation vs the sun's position.
  - Algorithms can be fine-tuned to move the panels continuously.

- ## Strings

  In Python, you can use either double quotes (") or single quotes (') to indicate a string value.

  - You just need to be consistent - always use the same type of quotes with the value.
  - For example, `"morning'` is **NOT** acceptable.
  - But you could use either `"morning"` or `'morning'`.

Goals:

- Assign the 🔧variable `state` as `'morning'`.
- Use an `if` statement with the 🔧condition `state == 'morning'`.
- Use two `elif` statements:
- One with the condition `state == 'afternoon'`.
- Another with the condition `state == 'night'`.

Tools Found: State, Variables, bool

Solution:

```
1   from codex import *
2   from time import sleep
3
4   # Constants for peripherals
5   PERIOD = 20                 # Milliseconds / Hz
6   CYCLE = 2**16 // PERIOD
7   FORWARD = 50                # Percent for 45 degree angle clockwise
8   CENTER = 75                 # Percent for 0 degree angle (centered)
9   BACKWARD = 100              # Percent for 45 degree angle counterclockwise
10
11  # Set up peripherals
12  panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
13  light_sensor = exp.analog_in(exp.PORT1)
14
15  # Set the servo by calculating the duty_cycle
16  def set_servo(percent):
17      return CYCLE * percent // 100
18
19  # Main program
20  panels.duty_cycle = set_servo(FORWARD)
21  state = 'morning'
22
23  while True:
24      if state == 'morning':
25          panels.duty_cycle = set_servo(FORWARD)
26      elif state == 'afternoon':
27          panels.duty_cycle = set_servo(CENTER)
28      elif state == 'night':
```

```
29          panels.duty_cycle = set_servo(BACKWARD)
30
31      if buttons.was_pressed(BTN_A):
32          break
33
34  # Stop the servo
35  panels.duty_cycle = 0
```

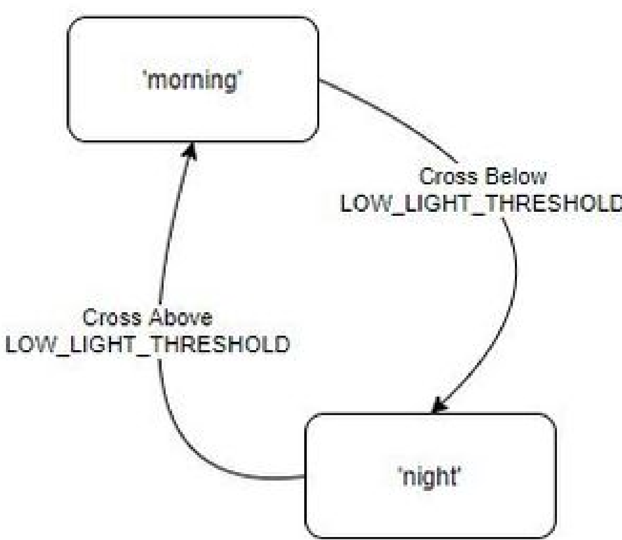
Objective 5 **- Morning and Night**

You have a system for a finite-state machine and a way to sense light, so let's put them to use.

Start by switching the state between `'morning'` and `'night'`.

- You can switch states in your `while True:` loop.
- You will ignore the `'afternoon'` state for now.
- You should use the finite-state machine diagram as a reference.

The finite-state machine for this project looks like this:



Your code needs to transition between the two states using a `LOW_LIGHT` threshold. Define a constant for the threshold based on your light sensor readings.

- Look at the results you wrote down from the console panel.
- Identify a low value threshold for the light sensor.
- During my testing, a low threshold is less than **15000**.

Define a constant for the low light threshold:

```
LOW_LIGHT = 12000
```

Now use an `if` statement with the current light sensor reading and `LOW_LIGHT` to transition from `'morning'` to `'night'` and from `'night'` to `'morning'`.

Check the 'Trek!

Follow the CodeTrek so the light sensor reading triggers the transitions.

Physical Interaction: *Try it*

Run the code.

- Start with the light sensor in daylight.
- Cover the light sensor to simulate the dark.

Do the panels rotate to the correct position each time?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20                # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  FORWARD = 50               # Percent for 45 degree angle clockwise
8  CENTER = 75                # Percent for 0 degree angle (centered)
9  BACKWARD = 100             # Percent for 45 degree angle counterclockwise
10 LOW_LIGHT = 12000          # Threshold ambient light for night-time
```

Define a constant for `LOW_LIGHT` that is the threshold for `'night'`.

- Assign the `LOW_LIGHT` value from your light sensor readings.

```
11
12 # Set up peripherals
13 panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
14 light_sensor = exp.analog_in(exp.PORT1)
15
16 # Set the servo by calculating the duty_cycle
17 def set_servo(percent):
18     return CYCLE * percent // 100
19
20 # Main program
21 panels.duty_cycle = set_servo(FORWARD)
22 state = 'morning'
23
24 while True:
25     if state == 'morning':
26         if light_sensor.value < LOW_LIGHT:
27             state = 'night'
28             panels.duty_cycle = set_servo(BACKWARD)
```

Modify the `'morning'` state by adding an `if` statement to compare the light sensor reading.

- If it is less than `LOW_LIGHT`, change the state to `'night'`.
- Set the panels to the `BACKWARD` position.

```
29     elif state == 'afternoon':
30         panels.duty_cycle = set_servo(CENTER)
```

Do not make any changes here at this time.

```
31     elif state == 'night':
32         if light_sensor.value > LOW_LIGHT:
33             state = 'morning'
34             panels.duty_cycle = set_servo(FORWARD)
```

Modify the `'night'` state by adding an `if` statement to compare the light sensor reading.

- If it is greater than `LOW_LIGHT`, change the state to `'morning'`.
- Set the panels to the `FORWARD` position.

```
35
36     if buttons.was_pressed(BTN_A):
37         break
38
39 # Stop the servo
40 panels.duty_cycle = 0
```

Goals:

- Define the constant `LOW_LIGHT`.
- Use an `if` statement with the condition `light_sensor.value < LOW_LIGHT`.
- Use an `if` statement with the condition `light_sensor.value > LOW_LIGHT`.
- Assign the variable `state` as `'night'`.

Tools Found: State, Constants, bool, Variables

Solution:

```
1  from codex import *
2  from time import sleep
3  
4  # Constants for peripherals
5  PERIOD = 20                 # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  FORWARD = 50                # Percent for 45 degree angle clockwise
8  CENTER = 75                 # Percent for 0 degree angle (centered)
9  BACKWARD = 100              # Percent for 45 degree angle counterclockwise
10 LOW_LIGHT = 12000           # Threshold ambient light for night-time
11 
12 # Set up peripherals
13 panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
14 light_sensor = exp.analog_in(exp.PORT1)
15 
16 # Set the servo by calculating the duty_cycle
17 def set_servo(percent):
18     return CYCLE * percent // 100
19 
20 # Main program
21 panels.duty_cycle = set_servo(FORWARD)
22 state = 'morning'
23 
24 while True:
25     if state == 'morning':
26         if light_sensor.value < LOW_LIGHT:
27             state = 'night'
28             panels.duty_cycle = set_servo(BACKWARD)
29     elif state == 'afternoon':
30         panels.duty_cycle = set_servo(CENTER)
31     elif state == 'night':
32         if light_sensor.value > LOW_LIGHT:
33             state = 'morning'
34             panels.duty_cycle = set_servo(FORWARD)
35 
36     if buttons.was_pressed(BTN_A):
37         break
38 
39 # Stop the servo
40 panels.duty_cycle = 0
```

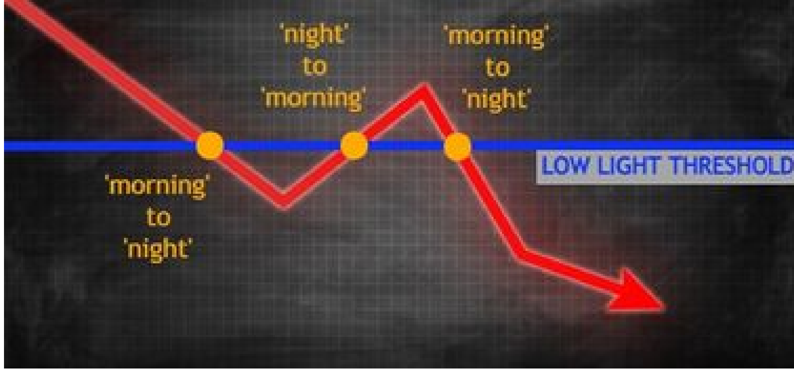
Objective 6 - Debouncing

There is a small problem with your state machine!

The state transitions rely on light values dropping or rising continuously. Unfortunately, real-world sensors are almost never perfect.

- Analog inputs can very easily **bounce** around.

Examine this chart:



The state is transitioning between 'morning' and 'night' at the `LOW_LIGHT` threshold.

But...it is transitioning twice instead of the expected single transition.

This is called bouncing. This could become an even **BIGGER** problem as you add more states to the system.

- You do not want your system inadvertently changing to an unexpected state!!

What can you do to solve this problem?

- You can just delay before taking the next reading.
- This will reduce the opportunity for bouncing around threshold values.

Check the 'Trek!

Add a simple delay to debounce your application's state transitions. A half-second should be enough!

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20                # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  FORWARD = 50               # Percent for 45 degree angle clockwise
8  CENTER = 75                # Percent for 0 degree angle (centered)
9  BACKWARD = 100             # Percent for 45 degree angle counterclockwise
10 LOW_LIGHT = 12000          # Threshold ambient light for night
11
12 # Set up peripherals
13 panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
14 light_sensor = exp.analog_in(exp.PORT1)
15
16 # Set the servo by calculating the duty_cycle
17 def set_servo(percent):
18     return CYCLE * percent // 100
19
20 # Main program
21 panels.duty_cycle = set_servo(FORWARD)
22 state = 'morning'
23
24 while True:
25     if state == 'morning':
26         if light_sensor.value < LOW_LIGHT:
27             state = 'night'
28             panels.duty_cycle = set_servo(BACKWARD)
29     elif state == 'afternoon':
30         panels.duty_cycle = set_servo(CENTER)
31     elif state == 'night':
32         if light_sensor.value > LOW_LIGHT:
33             state = 'morning'
34             panels.duty_cycle = set_servo(FORWARD)
```

```
35
36      sleep(0.5)
```

Add a short delay to debounce the state transitions.

- The delay can be a little longer or shorter than half a second.
- Watch your indenting! The delay is NOT part of the `if` statement block.

```
37
38      if buttons.was_pressed(BTN_A):
39          break
40
41  # Stop the servo
42  panels.duty_cycle = 0
```

Hint:

- **What are some events that could cause bouncing?**
  - A flicker in a lightbulb causing an inconsistent reading on a light sensor
  - A user pressing a button with varied pressure causing multiple "presses"
  - Electricity jumping between *closing* metal contacts
  - A user clicking a mouse multiple times accidentally

Goal:

- Call `sleep(0.5)`.

Tools Found: State, Debouncing, Timing

Solution:

```
1   from codex import *
2   from time import sleep
3
4   # Constants for peripherals
5   PERIOD = 20                # Milliseconds / Hz
6   CYCLE = 2**16 // PERIOD
7   FORWARD = 50               # Percent for 45 degree angle clockwise
8   CENTER = 75                # Percent for 0 degree angle (centered)
9   BACKWARD = 100             # Percent for 45 degree angle counterclockwise
10  LOW_LIGHT = 12000          # Threshold ambient light for night
11
12  # Set up peripherals
13  panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
14  light_sensor = exp.analog_in(exp.PORT1)
15
16  # Set the servo by calculating the duty_cycle
17  def set_servo(percent):
18      return CYCLE * percent // 100
19
20  # Main program
21  panels.duty_cycle = set_servo(FORWARD)
22  state = 'morning'
23
24  while True:
25      if state == 'morning':
26          if light_sensor.value < LOW_LIGHT:
27              state = 'night'
28              panels.duty_cycle = set_servo(BACKWARD)
29      elif state == 'afternoon':
30          panels.duty_cycle = set_servo(CENTER)
31      elif state == 'night':
32          if light_sensor.value > LOW_LIGHT:
33              state = 'morning'
```

```
34              panels.duty_cycle = set_servo(FORWARD)
35      # debounce
36      sleep(0.5)
37
38      if buttons.was_pressed(BTN_A):
39          break
40
41  # Stop the servo
42  panels.duty_cycle = 0
```

Quiz 2 - Check Your Understanding: Light Sensors

Question 1: What type of peripheral is the light sensor?

✓ Analog input

✗ Analog output

✗ Digital input

✗ Variable input

Question 2: In a light sensor reading, **MORE LIGHT** =

✓ Higher values

✗ Lower values

✗ Consistent values

✗ Inconsistent values

Question 3: A flickering light bulb could cause:

✓ Bouncing

✗ Transitioning

✗ An error

✗ The program to stop

Question 4: What is one way to debounce a state transition?

✓ Add a short delay between readings.

✗ Change the value of the constant.

✗ Use a different peripheral.

✗ Change the condition.

Objective 7 - Simulate the Sun

You need direct sunlight for the afternoon state.

Attach the white LED to simulate the sun!

Connect Peripheral

Connect the **white LED** to **PORT2** on your CodeX.

Now that you have "sunlight", use it to find the `HIGH_LIGHT` threshold.

- `print()` the light sensor readings to the **Console Panel** and observe the high values when the white LED is close to the sensor.

Check the Hints for the algorithm.

Check the 'Trek!

Follow the CodeTrek to set up the white LED as a sun simulator and read the light sensor to find the high light threshold.

Physical Interaction

Run the code. Move the white LED close to the light sensor to watch the values increase.

When the LED is close to the light sensor make note of the **highest** values.

- The values should go higher than **30000**
- This will be your high light threshold

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20                # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  FORWARD = 50               # Percent for 45 degree angle clockwise
8  CENTER = 75                # Percent for 0 degree angle (centered)
9  BACKWARD = 100             # Percent for 45 degree angle counterclockwise
10 LOW_LIGHT = 12000          # Threshold ambient light for night
11 #TODO: Define the constants `LED_ON` and `LED_OFF`
```

Define constants for the LED.

- `LED_ON` is `True`.
- `LED_OFF` is `False`.

```
12
13 # Set up peripherals
14 panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
15 light_sensor = exp.analog_in(exp.PORT1)
16 led = exp.digital_out(exp.PORT2)
```

Set up the LED on PORT2.

- The LED is a digital output peripheral.
- It will not use PWM for this mission.

```
17
18 # Set the servo by calculating the duty_cycle
19 def set_servo(percent):
20     return CYCLE * percent // 100
21
22 def set_led(val):
23     #TODO: set the value of the LED
```

Define the function that turns on/off the LED.

- `led.value = val`

```
24
25 # Main program
26 panels.duty_cycle = set_servo(FORWARD)
27 state = 'morning'
28 set_led(LED_ON)
```

Turn on the LED to simulate the sun.

- The LED will stay on during program execution.
- You will physically move it closer and farther from the light sensor.

```
29
30 while True:
31     print("Light sensor: ", light_sensor.value)
```

Read the light sensor and `print()` its values to the **Console Panel**.

- Observe the readings when the "sun" is overhead.
- Make a note of the high light threshold.

```
32
33     if state == 'morning':
34         if light_sensor.value < LOW_LIGHT:
35             state = 'night'
36             panels.duty_cycle = set_servo(BACKWARD)
37     elif state == 'afternoon':
38         panels.duty_cycle = set_servo(CENTER)
39     elif state == 'night':
40         if light_sensor.value > LOW_LIGHT:
41             state = 'morning'
42             panels.duty_cycle = set_servo(FORWARD)
43
44     sleep(0.5)
45
46     if buttons.was_pressed(BTN_A):
47         break
48
49 # Stop the servo and LED
50 panels.duty_cycle = 0
51 #TODO: Turn off the LED
```

Turn off the LED when the program ends.

Hint:

- ## Sun Simulation Algorithm

  You are adding a digital input peripheral to this project. It is one you are familiar with.

  - Set up the white LED on PORT 2.
  - Define `LED_ON` and `LED_OFF` constants for the LED.

  - Define a function to turn on/off the LED.
  - Turn **ON** the LED in the main program.
  - `print()` the light sensor reading to the **Console Panel**.

How much of this can you do without checking the CodeTrek?

Goals:

- Define the 🔧constants `LED_ON` and `LED_OFF`.
- Define the 🔧function `set_led(val)`.
- Assign `led.value` as `val`.
- Call `set_led()` at least twice.
- 🔧Print `light_sensor.value`.

Tools Found: Constants, Functions, Print Function

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Constants for peripherals
5  PERIOD = 20                # Milliseconds / Hz
6  CYCLE = 2**16 // PERIOD
7  FORWARD = 50               # Percent for 45 degree angle clockwise
8  CENTER = 75                # Percent for 0 degree angle (centered)
9  BACKWARD = 100             # Percent for 45 degree angle counterclockwise
10 LOW_LIGHT = 12000          # Threshold ambient light for night
11 LED_ON = True
12 LED_OFF = False
13
14
15 # Set up peripherals
16 panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
17 light_sensor = exp.analog_in(exp.PORT1)
18 led = exp.digital_out(exp.PORT2)
19
20 # Set the servo by calculating the duty_cycle
21 def set_servo(percent):
22     return CYCLE * percent // 100
23
24 def set_led(val):
25     led.value = val
26
27 # Main program
28 panels.duty_cycle = set_servo(FORWARD)
29 state = 'morning'
30 set_led(LED_ON)
31
32 while True:
33     print("Light sensor: ", light_sensor.value)
34
35     if state == 'morning':
36         if light_sensor.value < LOW_LIGHT:
37             state = 'night'
38             panels.duty_cycle = set_servo(BACKWARD)
39     elif state == 'afternoon':
40         panels.duty_cycle = set_servo(CENTER)
41     elif state == 'night':
42         if light_sensor.value > LOW_LIGHT:
43             state = 'morning'
44             panels.duty_cycle = set_servo(FORWARD)
45     # debounce
46     sleep(0.5)
47
48     if buttons.was_pressed(BTN_A):
```

```
49          break
50
51  # Stop the servo and LED
52  panels.duty_cycle = 0
53  set_led(LED_OFF)
```

Objective 8 **- Automatic Sun Tracking**

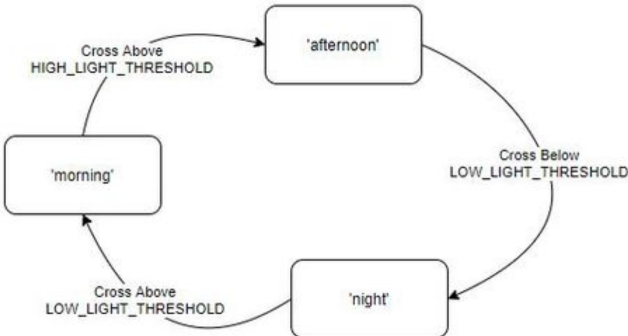
Add the final state to your code!

- `'afternoon'` is the final state for the solar tracker

The transition to the `'afternoon'` state should occur when the sun is at its **brightest**.

- You should now have a value for the `HIGH_LIGHT` threshold, from your observations during the last Objective.
- Define a constant for `HIGH_LIGHT` using your high value readings.
- During my testing, a high threshold is more than **30000**.

The finite-state machine looks like this:



- The transition from `'morning'` to `'afternoon'` should utilize the new `HIGH_LIGHT` threshold.
- And the transition from `'afternoon'` to `'night'` will utilize the `LOW_LIGHT` threshold.

Check the 'Trek!

Follow the CodeTrek to complete the finite-state machine.

- Use the constants `LOW_LIGHT` and `HIGH_LIGHT` with the light sensor readings to transition from one state to the next state.

Physical Interaction

1. Place the white LED next to the light sensor to transition from `'morning'` to `'afternoon'`
2. Cover the light sensor to transition from `'afternoon'` to `'night'`
3. Uncover the light sensor to transition from `'night'` to `'morning'`

And... **REPEAT**

CodeTrek:

```
1   from codex import *
2   from time import sleep
3
4   # Constants for peripherals
5   PERIOD = 20                # Milliseconds / Hz
6   CYCLE = 2**16 // PERIOD
7   FORWARD = 50               # Percent for 45 degree angle clockwise
8   CENTER = 75                # Percent for 0 degree angle (centered)
9   BACKWARD = 100             # Percent for 45 degree angle counterclockwise
10  LOW_LIGHT = 12000          # Threshold ambient light for night-time
11  HIGH_LIGHT = 30000         # Threshold ambient light for day-time
```

Define `HIGH_LIGHT` with the threshold for bright sunlight.

- Use the value from your observations for this constant.

```
12 LED_ON = True
13 LED_OFF = False
14
15 # Set up peripherals
16 panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
17 light_sensor = exp.analog_in(exp.PORT1)
18 led = exp.digital_out(exp.PORT2)
19
20 # Set the servo by calculating the duty_cycle
21 def set_servo(percent):
22     return CYCLE * percent // 100
23
24 def set_led(val):
25     led.value = val
26
27 # Main program
28 panels.duty_cycle = set_servo(FORWARD)
29 state = 'morning'
30 set_led(LED_ON)
31
32 while True:
33     print("Light sensor: ", light_sensor.value)
```

The `print()` statement was used for testing.

- You can keep it in the code or delete it.

```
34
35     if state == 'morning':
36         if light_sensor.value > HIGH_LIGHT:
37             state = 'afternoon'
38             panels.duty_cycle = set_servo(CENTER)
```

Modify the code for the `'morning'` state to transition to `'afternoon'`.

- Use the `HIGH_LIGHT` threshold.
- If `True` change the state and set the panels to `CENTER`.

```
39     elif state == 'afternoon':
40         if light_sensor.value < LOW_LIGHT:
41             state = 'night'
42             panels.duty_cycle = set_servo(BACKWARD)
```

Modify the code for the `'afternoon'` state to transition to `'night'`.

- Use the `LOW_LIGHT` threshold.
- If `True` change the state and set the panels to `BACKWARD`.

```
43     elif state == 'night':
44         if light_sensor.value > LOW_LIGHT:
45             state = 'morning'
46             panels.duty_cycle = set_servo(FORWARD)
```

The code for the `'night'` state does not need to change.

```
47
48     sleep(0.5)
49
50     if buttons.was_pressed(BTN_A):
51         break
52
53 # Stop the servo and LED
```

```
54  panels.duty_cycle = 0
55  set_led(LED_OFF)
```

Hint:

- ## Sunlight Simulation

  Can you think of other ways to simulate sunlight?

  - The LED can use PWM and the potentiometer readings to change from dim to bright.
  - You can use a timer or switch to turn on/off the LED.
  - A servo can spin the LED around the light sensor.

  With the peripherals kit and your imagination, there are many possibilities!

Goals:

- Define the constant `HIGH_LIGHT`.
- Assign the variable `state` at least four times.
- Assign the variable `state` as `'afternoon'`.

Tools Found: State, Constants, Variables

Solution:

```
1   from codex import *
2   from time import sleep
3
4   # Constants for peripherals
5   PERIOD = 20                 # Milliseconds / Hz
6   CYCLE = 2**16 // PERIOD
7   FORWARD = 50                # Percent for 45 degree angle clockwise
8   CENTER = 75                 # Percent for 0 degree angle (centered)
9   BACKWARD = 100              # Percent for 45 degree angle counterclockwise
10  LOW_LIGHT = 12000           # Threshold ambient light for night-time
11  HIGH_LIGHT = 30000          # Threshold ambient light for day-time
12  LED_ON = True
13  LED_OFF = False
14
15  # Set up peripherals
16  panels = exp.pwm_out(exp.PORT0, frequency=PERIOD) # 180 servo
17  light_sensor = exp.analog_in(exp.PORT1)
18  led = exp.digital_out(exp.PORT2)
19
20  # Set the servo by calculating the duty_cycle
21  def set_servo(percent):
22      return CYCLE * percent // 100
23
24  def set_led(val):
25      led.value = val
26
27  # Main program
28  panels.duty_cycle = set_servo(FORWARD)
29  state = 'morning'
30  set_led(LED_ON)
31
32  while True:
33      # Optional - can be deleted
34      print("Light sensor: ", light_sensor.value)
35
36      if state == 'morning':
37          if light_sensor.value > HIGH_LIGHT:
38              state = 'afternoon'
39              panels.duty_cycle = set_servo(CENTER)
40      elif state == 'afternoon':
41          if light_sensor.value < LOW_LIGHT:
```

```
42              state = 'night'
43              panels.duty_cycle = set_servo(BACKWARD)
44      elif state == 'night':
45          if light_sensor.value > LOW_LIGHT:
46              state = 'morning'
47              panels.duty_cycle = set_servo(FORWARD)
48
49      sleep(0.5)
50
51      if buttons.was_pressed(BTN_A):
52          break
53
54  # Stop the servo and LED
55  panels.duty_cycle = 0
56  set_led(LED_OFF)
```

Mission 7 Complete

You've completed project Solar Tracking!

You now know how to use positional servos and apply a more complex 🔧state machine.

The ship's power supply has been boosted over 30% by rotating the solar panels!

You have created enough power to get the crew to Mars!!

Mission 8 - Prepare Lander!

This project provides a tutorial on the object sensor and ties in multiple different peripherals!

Mission Briefing:

The crew has survived their long trip to Mars.

- Now they need to enter the atmosphere and land safely on the surface.

Your task is to incorporate a few systems to prepare the lander, sense the surface, and provide notifications!

You can make the crew's first experience on Mars a great one with a smooth landing sequence!

Project: Prepare Lander will guide you through preparing the lander for atmospheric entry and landing.

Project Goals:

- Learn about the Object Sensor
- Tie in together a servo and the NeoPixels in a landing system
- Create a simple state machine
- Ease the crew's entry sequence for landing on Mars!!

Note:

You will need extra materials for this mission:

- A small phillips screwdriver
- A small ruler or measuring device

Objective 1 - Plan of Attack

Project: Prepare Lander will require many different components!

You should start by taking a look at the different phases (or states) of the landing on Mars:

The first phase - "Init" phase

- In this phase the lander will be about to enter the Mars Atmosphere
- The crew will be notified with a yellow light
- The landing gear will be completely retracted into the craft

The second phase - "Prepare" phase

- This phase will be entered when the craft detects the ground
- The crew will be notified with a red light
- The landing gear will be extended and ready for landing

The third phase - "Landed" phase

- This will be entered when the craft touches down on the surface
- The crew will be notified with a green light

©2024 Firia Labs

Project: Prepare Lander

The Mars lander will be able to:

- Display the landing state to the crew with lights.
- Sense the surface and extend the landing gear.
- Detect touchdown on the planet.

Create a New File!

Use the **File → New File** menu to create a new file called ***PrepareLander***.

Goal:

- Create a new file named `PrepareLander`.

Solution:

N/A

Objective 2 **- Crew Display!**

You will start by setting up a display for the crew.

The NeoPixel ring will make an excellent display.

- You can use the multi-color display to show different Lander States!

Connect Peripheral

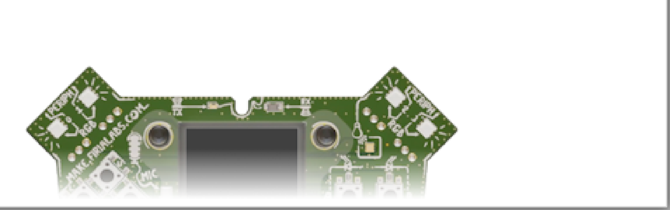
Connect the **8 RGB NeoPixel ring** to **PORT0** on your CodeX. Use the three loose wires provided in the peripheral kit.

The back side of the NeoPixel ring has three labels:

- **G** or *ground* is connected to the **BLACK** wire.
- **V** or *voltage* is connected to the **ORANGE** wire.
- **DI** or *data input* is connected to the **YELLOW** wire.

  On the CodeX, the YELLOW wire goes to **S**, the ORANGE in the middle, and the BLACK to the **G**. The colors of your wires may be different. That is okay. Just connect the wires to the correct pins.

Disconnect all other peripherals from the CodeX.



Add code to light up the NeoPixels. To prepare the display for the crew, you will need to:

- Set up the NeoPixel ring.
- Define constants for the colors YELLOW, RED, and GREEN.
- Define a function that lights all pixels a color.

Check the 'Trek!

Follow the CodeTrek to prepare the display for the crew.

Run It!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  #TODO: Define RGB_YELLOW, RGB_RED, and RGB_GREEN
```

Define constants for the colors the ring will use:

- `RGB_YELLOW = (20, 20, 0)`
- `RGB_RED = (20, 0, 0)`
- `RGB_GREEN = (0, 20, 0)`

```
6
7  # Set up the peripherals
8  power.enable_periph_vcc(True)
9  np = neopixel.NeoPixel(exp.PORT0, 8)
```

Set up the NeoPixel ring on PORT0.

- It has 8 pixels on the ring.
- Manually turn on power to the ring.

```
10
11 # Set all pixels to one color
12 def set_lighting(rgb_color):
13     for pixel in range(8):
14         np[pixel] = rgb_color
```

Define a function for turning on all pixels the same color.

- This is the same function you defined in **HatchLock**, but with a descriptive name for this mission.

```
15
16 # Main program
17 set_lighting(RGB_YELLOW)
18 sleep(3)
19 set_lighting((0, 0 , 0))
```

Test your code and make sure it works.

- Set all pixels to RGB_YELLOW.

- After a delay, turn all pixels OFF.

Hints:

- ## NeoPixel Ring Code

  The code for setting up the NeoPixel ring is very similar to the beginning of the program for **HatchLock**. You may want to review your code.

- ## Additional Power

  REMINDER: The NeoPixel ring requires a lot of power, so you will manually turn the power on. Use this code to provide a boost of power:

```
power.enable_periph_vcc(True)
```

Goals:

- Call `power.enable_periph_vcc(True)`.
- Define the constants `RGB_YELLOW`, `RGB_RED`, and `RGB_GREEN`.
- Define the function `set_lighting(rgb_color)`.
- Call `set_lighting()` at least twice.

Tools Found: Constants, Functions

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
6  RGB_RED = (20, 0, 0)
7  RGB_GREEN = (0, 20, 0)
8
9  # Set up the peripherals
10 power.enable_periph_vcc(True)
11 np = neopixel.NeoPixel(exp.PORT0, 8)
12
13 # Set all pixels to one color
14 def set_lighting(rgb_color):
15     for pixel in range(8):
16         np[pixel] = rgb_color
17
18 # Main program
19 set_lighting(RGB_YELLOW)
20 sleep(3)
21 set_lighting((0, 0 , 0))
```

Objective 3 **- Connect the Object Sensor!**

To enter Phase II, the lander craft needs to detect the ground.

Use the object sensor to detect the ground.

The **object sensor** is a peripheral that:

- Emits an IR (Infrared) light.
- Detects the reflected IR energy with a phototransistor.
- Outputs a digital `True` (HIGH) for ***detected*** or `False` (LOW) for ***not detected***.
- Check the Hints for more details about the **object sensor**.

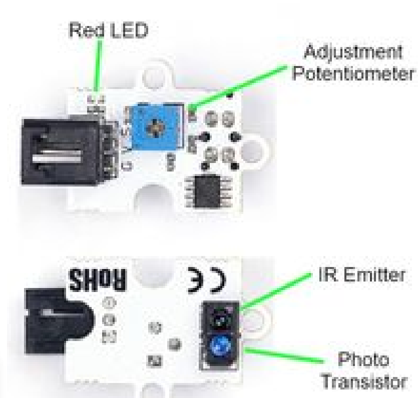
Connect Peripheral

Connect the **object sensor** to **PORT1** on your CodeX.

Set the ideal distance for the object sensor!

The object sensor contains a small blue potentiometer on the top side.

- This has a white phillips (cross) screw inside.
- Adjusting the screw will make the sensor more or less "sensitive".
- Adjusting the object sensor:
  - Full Clockwise = Not Sensitive Enough
  - Full Counterclockwise = Too Sensitive



The goal is to adjust to the correct distance.

- The ideal distance for the object sensor is 2.5 millimeters.
- This distance was determined by the chip manufacturer.

Physical Interaction

1. With a small phillips screwdriver, adjust the screw halfway between full clockwise and full counterclockwise position.
2. Hold the sensor above a white surface (with the blue potentiometer facing up and the IR sensor facing down).
3. Move the sensor slowly up and down. Observe the red LED next to the potentiometer. It turns **ON** when *no object is detected* and **OFF** when *an object is detected*.
4. Hold the sensor at 2.5 millimeters.
5. Adjust the phillips screw until the red LED turns **OFF** precisely at the threshold of 2.5 millimeters.

Add some code to read the digital output of the object sensor!

Your algorithm for reading the object sensor looks like this:

- Set up the object sensor.
- Define a constant for `GROUND_DETECTED`.
- Use a `while True:` loop to read the object sensor continuously.
- `if` the object sensor **detects** the ground, turn the NeoPixels **GREEN**.
- Otherwise turn the NeoPixels **YELLOW**.

Check the 'Trek!

Follow the CodeTrek to use the object sensor to control the display for the crew.

Physical Interaction

Run the code.

- Start with the object sensor higher than 2.5 millimeters.
- Lower the object sensor until ground is detected.
- The NeoPixel ring should turn from **YELLOW** to **GREEN**.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
6  RBG_RED = (20, 0, 0)
7  RGB_GREEN = (0, 20, 0)
8  GROUND_DETECTED = True
```

Define a constant for the object sensor.

```
9
10 # Set up the peripherals
11 power.enable_periph_vcc(True)
12 np = neopixel.NeoPixel(exp.PORT0, 8)
13 ground_sensor = exp.digital_in(exp.PORT1)
```

Set up the object sensor on PORT1.

- Name the variable `ground_sensor`.
- Tell the CodeX a digital input peripheral is connected on PORT1.

```
14
15 # Set all pixels to one color
16 def set_lighting(rgb_color):
17     for pixel in range(8):
18         np[pixel] = rgb_color
19
20 # Main program
21 while True:
22     if ground_sensor.value == GROUND_DETECTED:
23         set_lighting(RGB_GREEN)
24     else:
25         set_lighting(RGB_YELLOW)
```

- **Remember:** `ground_sensor.value` is `True` when it detects something.

```
26
```

Hint:

- ## Object Sensors

  The **Object Sensor** is sometimes called a **Hunt Sensor**. In this project you will call it a **Ground Sensor**.

It can be used to:

- Detect nearby obstacles.
- Distinguish between black and white colors (Line Sensing).
- Count pulses as an object passes by (Wheel Encoders).

Goals:

- Set up the **object sensor** by assigning the variable `ground_sensor` as the output of `exp.digital_in(exp.PORT1)`.
- Define the constant `GROUND_DETECTED`.
- Use an `if` statement with the condition `ground_sensor.value == GROUND_DETECTED`.

- Call `set_lighting(RGB_GREEN)` and `set_lighting(RGB_YELLOW)`.

Tools Found: CPU and Peripherals, Variables, Constants, bool

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
6  RGB_RED = (20, 0, 0)
7  RGB_GREEN = (0, 20, 0)
8  GROUND_DETECTED = True
9
10 # Set up the peripherals
11 power.enable_periph_vcc(True)
12 np = neopixel.NeoPixel(exp.PORT0, 8)
13 ground_sensor = exp.digital_in(exp.PORT1)
14
15 # Set all pixels to one color
16 def set_lighting(rgb_color):
17     for pixel in range(8):
18         np[pixel] = rgb_color
19
20 # Main program
21 while True:
22     if ground_sensor.value == GROUND_DETECTED:
23         set_lighting(RGB_GREEN)
24     else:
25         set_lighting(RGB_YELLOW)
26
27
```

Objective 4 - Pull Up!

Why was the object sensor not providing the right value?

When the object sensor was very close to the white surface, you probably noticed that it stopped detecting and the red LED turned on.

The object sensor has a very weak boolean **HIGH** value.

- It is necessary to give the input a little boost.
- You can boost the level by changing the default setting of `pull` when you set up the sensor.

Concept

What is a **pull** on an input pin?

- The pull determines the default value of a pin when nothing is connected.
- Pull values are very weak and can be overcome by applying an external HIGH or LOW voltage.

The pull of a pin can be set to:

- Pull.UP = weak pull toward 3 Volts (HIGH Logic)
- Pull.DOWN = weak pull toward 0 Volts (LOW Logic)
- None = the pin can drift between high and low

The pins on the CodeX default to `None`.

Check the 'Trek!

Follow the CodeTrek to set the **pull** of the object sensor.

Physical Interaction

Run the code. Move the object sensor closer and further from a surface.

- Does the sensor continue to detect an object at very close range?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
6  RGB_RED = (20, 0, 0)
7  RGB_GREEN = (0, 20, 0)
8  GROUND_DETECTED = True
9
10 # Set up the peripherals
11 power.enable_periph_vcc(True)
12 np = neopixel.NeoPixel(exp.PORT0, 8)
13 ground_sensor = exp.digital_in(exp.PORT1, pull=digitalio.Pull.UP)
```

Set the **pull** for the object sensor to `digitalio.Pull.UP`.

```
14
15 # Set all pixels to one color
16 def set_lighting(rgb_color):
17     for pixel in range(8):
18         np[pixel] = rgb_color
19
20 # Main program
21 while True:
22     if ground_sensor.value == GROUND_DETECTED:
23         set_lighting(RGB_GREEN)
24     else:
25         set_lighting(RGB_YELLOW)
```

Hint:

- ## Sensor Pull

  The downside to using a pull is the external source has to be able to overcome the **strength** of the pull on that pin.

  - Most devices that require a pull up can easily overcome that by dragging the voltage down to 0.

  NOTE the buttons in the peripheral kit all default to `None` and do not require setting a value on the CodeX.

  - Other buttons or sensors may require some external (or internal) pull direction.

  Many peripherals in the real-world require you to use pulls on your input pins!

Goal:

- Set up the **pull** for the **object sensor** by assigning the variable `ground_sensor` as the output of `exp.digital_in(exp.PORT1, pull=digitalio.Pull.UP)`.

Tools Found: bool, Pins, Variables

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
```

```
 6  RGB_RED = (20, 0, 0)
 7  RGB_GREEN = (0, 20, 0)
 8  GROUND_DETECTED = True
 9
10  # Set up the peripherals
11  power.enable_periph_vcc(True)
12  np = neopixel.NeoPixel(exp.PORT0, 8)
13  ground_sensor = exp.digital_in(exp.PORT1, pull=digitalio.Pull.UP)
14
15  # Set all pixels to one color
16  def set_lighting(rgb_color):
17      for pixel in range(8):
18          np[pixel] = rgb_color
19
20  # Main program
21  while True:
22      if ground_sensor.value == GROUND_DETECTED:
23          set_lighting(RGB_GREEN)
24      else:
25          set_lighting(RGB_YELLOW)
26
```

Quiz 1 - Check Your Understanding: Object Sensors

Question 1: What type of peripheral is the object sensor?

✓ Digital input

✗ Digital output

✗ Analog input

✗ Analog output

Question 2: What values does the object sensor return?

✓ `True` for detected, `False` for not detected

✗ `False` for detected, `True` for not detected

✗ `1` for detected, `0` for not detected

✗ An integer between 0-65535

Question 3: What is one thing the object sensor CANNOT detect?

✓ Motion 10 feet away

✗ Close obstacles

✗ A black line on a white background

✗ Pulses as an object passes by

Question 4: What is the ideal distance to adjust the object sensor to?

✓ 2.5 mm

✗ 5 mm

✗ 2.5 cm

✗ 1 inch

Question 5: How can you boost the input of an object sensor?

✓ Change the default **pull** to `Pull.UP`

✗ Change the default **pull** to `Pull.DOWN`

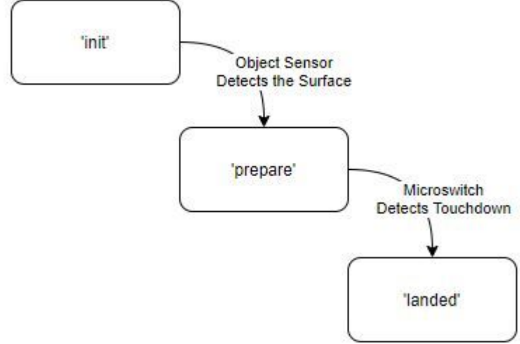
✗ Change the default **pull** to `None`

✗ Use the potentiometer.

Objective 5 **- Lander States**

Time to apply a finite-state machine to the lander!

The lander states should mimic the phases of landing almost identically.



This state diagram is pretty simple... *Why would I want to add states to my system? Wouldn't that just make the code more complex?*

Say as a remix idea you wanted to:

- Use a button to abort the landing and transition from the `'prepare'` state back to the `'init'` state?
- Add a fourth state before the `'init'` state that required some crew action to transition?

In this project, you will utilize states to avoid extra processor effort!

Let's write an algorithm from the diagram.

- Create a `lander_state` variable and initialize it to `'init'`.
- Set the lighting for the `'init'` state to `RGB_YELLOW`.
- Perform sensor checks for the appropriate sensors in your `while` loop based on your current state.
- Set the lighting for the `'prepare'` state to `RGB_RED`.
- For now, just immediately transition from the `'prepare'` state to the `'landed'` state.
  - You will add a touchdown sensor later!
- Set the lighting for the `'landed'` state to `RGB_GREEN`.

Check the 'Trek!

Follow the CodeTrek to set up the finite-state machine for landing the craft.

Physical Interaction

Run the code.

- Start with the object sensor higher than a surface.
- Gradually move the object sensor close to the surface.
- Your display should go from `'prepare'` to `'landed'`, or **yellow** lights to **green** lights.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
6  RGB_RED = (20, 0, 0)
7  RGB_GREEN = (0, 20, 0)
8  GROUND_DETECTED = True
9
10 # Set up the peripherals
11 power.enable_periph_vcc(True)
12 np = neopixel.NeoPixel(exp.PORT0, 8)
13 ground_sensor = exp.digital_in(exp.PORT1)
14
15 # Set all pixels to one color
16 def set_lighting(rgb_color):
17     for pixel in range(8):
18         np[pixel] = rgb_color
19
20 # Main program
21 # Initialize the first state and let object sensor settle
22 lander_state = 'init'
23 set_lighting(RGB_YELLOW)
24 sleep(1)
```

Initialize the finite-state machine.

- Set `lander_state` to `'init'`.
- Set the lights to `RGB_YELLOW`.
- Add a short delay so the object sensor can settle (debounce).

```
25
26 while True:
27     if lander_state == 'init':
28         if ground_sensor.value == GROUND_DETECTED:
29             lander_state = 'prepare'
30             set_lighting(RGB_RED)
```

First phase of the finite-state machine.

- If the state is `'init'`, then check the object sensor.
- If `GROUND_DETECTED`:
  - update the state variable to the next phase.
  - set the lights to `RGB_RED`.

```
31     elif lander_state == 'prepare':
32         lander_state = 'landed'
33         set_lighting(RGB_GREEN)
```

Second phase of the finite-state machine.

- If the state is `'prepare'`:
  - update the state variable to the next phase.
  - set the lights to `RGB_GREEN`.

NOTE: *This phase will be modified.*

```
34     elif lander_state == 'landed':
35         break
```

Third phase of the finite-state machine.

- If the state is `'landed'`, then end the program.
  - *Successful landing!*

```
36
```

Hint:

- ## Checking Sensors

  You will only check the sensors that are meaningful in your current 🔧state.

  For example:

  - You will only check a touchdown sensor when you are in the `'prepare'` state.
    - The touchdown sensor would not have any meaning in the `'init'` state.
  - If you had 100 touchdown sensors:
    - You wouldn't want to be checking those sensors constantly on your entire trip from Earth.

Goals:

- Assign the 🔧variable `lander_state` as `'init'`, `'prepare'`, and `'landed'`.
- Call `set_lighting(RGB_RED)`.
- Use an `if` statement with the 🔧condition `lander_state == 'init'`.
- Use an `elif` with the condition `lander_state == 'landed'`.

Tools Found: State, Loops, Variables, bool

Solution:

```
1  from codex import *
2  from time import sleep
3  
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
6  RGB_RED = (20, 0, 0)
7  RGB_GREEN = (0, 20, 0)
8  GROUND_DETECTED = True
9  
10 # Set up the peripherals
11 power.enable_periph_vcc(True)
12 np = neopixel.NeoPixel(exp.PORT0, 8)
13 ground_sensor = exp.digital_in(exp.PORT1)
14 
15 # Set all pixels to one color
16 def set_lighting(rgb_color):
17     for pixel in range(8):
18         np[pixel] = rgb_color
19 
20 # Main program
21 # Initialize the first state and let object sensor settle
22 lander_state = 'init'
23 set_lighting(RGB_YELLOW)
24 sleep(1)
25 
26 while True:
27     if lander_state == 'init':
28         if ground_sensor.value == GROUND_DETECTED:
29             lander_state = 'prepare'
30             set_lighting(RGB_RED)
31     elif lander_state == 'prepare':
32         lander_state = 'landed'
33         set_lighting(RGB_GREEN)
34     elif lander_state == 'landed':
35         break
36 
```

***Objective 6* - Sense Touchdown!**

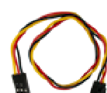
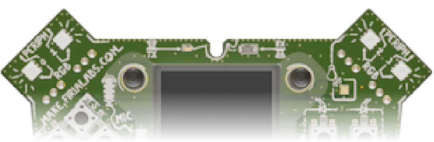
The Lander needs to know when it hits the surface!

You can use the microswitch as your touchdown sensor!

- The microswitch could be placed in hundreds of locations to detect impact with the surface
- In a safety critical application you might want multiple different microswitches!

Connect Peripheral

Connect the **microswitch** to **PORT2** on your CodeX.

You will read the microswitch in the second phase to detect a landing.

- Remember, the microswitch is `False` when pressed.
- So, define a constant with this value.

Now modify the second phase to read the switch and prepare for landing!

Check the 'Trek!

Follow the CodeTrek to complete the second phase of the finite-state machine.

Physical Interaction: *Try It!*

Run the code. Transition between the three states:

1. Hold the object sensor well above the surface to stay in the `'init'` state.
2. Move the object sensor toward the surface until it senses the ground to enter the `'prepare'` state.
3. Press the microswitch to the surface to sense touchdown and transition to the `'landed'` state!!

CodeTrek:

```
1 from codex import *
2 from time import sleep
3
4 # Define constants for the peripherals
5 RGB_YELLOW = (20, 20, 0)
6 RGB_RED = (20, 0, 0)
7 RGB_GREEN = (0, 20, 0)
8 GROUND_DETECTED = True       # Object sensor
9 TOUCHDOWN_DETECTED = False   # Microswitch
```

Define a constant for the microswitch.

- Since the microswitch is `False` when **pressed**, the value of the constant is `False`.

```
10
11  # Set up the peripherals
12  power.enable_periph_vcc(True)
13  np = neopixel.NeoPixel(exp.PORT0, 8)
14  ground_sensor = exp.digital_in(exp.PORT1)     # Object sensor
15  touchdown_sensor = exp.digital_in(exp.PORT2)  # Microswitch
```

Set up the microswitch on PORT2.

- Define the variable as `touchdown_sensor`.
- Tell the CodeX a digital input peripheral is connected on PORT2.

```
16
17  # Set all pixels to one color
18  def set_lighting(rgb_color):
19      for pixel in range(8):
20          np[pixel] = rgb_color
21
22  # Main program
23  # Initialize the first state and let object sensor settle
24  lander_state = 'init'
25  set_lighting(RGB_YELLOW)
26  sleep(1)
27
28  while True:
29      if lander_state == 'init':
30          # Read the object sensor
31          if ground_sensor.value == GROUND_DETECTED:
32              lander_state = 'prepare'
33              set_lighting(RGB_RED)
34      elif lander_state == 'prepare':
35          # Read the microswitch
36          if touchdown_sensor.value == TOUCHDOWN_DETECTED:
37              lander_state = 'landed'
38              set_lighting(RGB_GREEN)
```

Update the second phase to read the microswitch.

- If `TOUCHDOWN_DETECTED`:
  - update the state variable.
  - set the lights to `RGB_GREEN`.

```
39      elif lander_state == 'landed':
40          break
```

Hint:

- ## Turn Off the Lights!

  The power to the peripherals keeps the LEDs on, even after the program ends.

  If you want to turn the LEDs off, you can add code **AFTER** the `while True:` loop.

  ```
  ## turn off lights
  sleep(3)
  set_lighting((0, 0, 0))
  ```

  Remember: This code is **AFTER** the loop, so watch your indenting!

Goals:

- Set up the **microswitch** by assigning the 🔧variable `touchdown_sensor` as the output of `exp.digital_in(exp.PORT2)`.
- Define the 🔧constant `TOUCHDOWN_DETECTED`.
- Use an `if` statement with the 🔧condition `touchdown_sensor.value == TOUCHDOWN_DETECTED`.

Tools Found: Variables, Constants, bool

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
6  RGB_RED = (20, 0, 0)
7  RGB_GREEN = (0, 20, 0)
8  GROUND_DETECTED = True       # Object sensor
9  TOUCHDOWN_DETECTED = False   # Microswitch
10
11 # Set up the peripherals
12 power.enable_periph_vcc(True)
13 np = neopixel.NeoPixel(exp.PORT0, 8)
14 ground_sensor = exp.digital_in(exp.PORT1)     # Object sensor
15 touchdown_sensor = exp.digital_in(exp.PORT2)  # Microswitch
16
17 # Set all pixels to one color
18 def set_lighting(rgb_color):
19     for pixel in range(8):
20         np[pixel] = rgb_color
21
22 # Main program
23 # Initialize the first state and let object sensor settle
24 lander_state = 'init'
25 set_lighting(RGB_YELLOW)
26 sleep(1)
27
28 while True:
29     if lander_state == 'init':
30         # Read the object sensor
31         if ground_sensor.value == GROUND_DETECTED:
32             lander_state = 'prepare'
33             set_lighting(RGB_RED)
34     elif lander_state == 'prepare':
35         # Read the microswitch
36         if touchdown_sensor.value == TOUCHDOWN_DETECTED:
37             lander_state = 'landed'
38             set_lighting(RGB_GREEN)
39     elif lander_state == 'landed':
40         break
```

Objective 7 - Set up the Landing Gear!

Connect the landing gear to your lander!

The positional (180) servo is the right tool for moving a landing gear.

You want the landing gear to:

- Move to a known position (either **extended** or **retracted**)
- Stay in position once it has moved

For a bit of fun, you could tape a picture of landing gear to the positional servo so that you can see them move!!

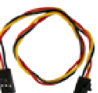
Connect Peripheral

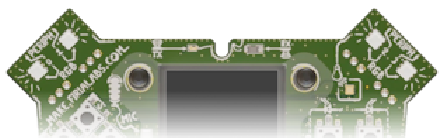
Connect the **180 servo** to **PORT3** on your CodeX.

- Attach **one** *servo-horn* to the servo.

NOTE: Your servo will have orange, red, and brown wires.

- The **ORANGE** wire is inserted into the **S**
- The **BROWN** wire is inserted into the **G**





The landing gear should start in the **retracted** position when the lander is the `'init'` state.

- It should only be **extended** when you enter the `'prepare'` state.
- This will keep the gear from dragging through the atmosphere on reentry!

Define a function to set the servo to the desired position.

- You will use many of the constants defined in **Project: Solar Tracking**.
- The function `set_servo()` is the same one in **Project: Solar Tracking**

Check the 'Trek!

Follow the CodeTrek to set up the servo and use it to control the landing gear.

Physical Interaction: *Try It!*

Run the code. Transition between the three states:

1. Hold the object sensor well above the surface to stay in the `'init'` state.
2. Move the object sensor toward the surface until it senses the ground to enter the `'prepare'` state.
3. Press the microswitch to the surface to sense touchdown and transition to the `'landed'` state!!

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
6  RGB_RED = (20, 0, 0)
7  RGB_GREEN = (0, 20, 0)
8  GROUND_DETECTED = True        # Object sensor
9  TOUCHDOWN_DETECTED = False    # Microswitch
10 PERIOD = 20                   # Milliseconds / Hz
11 CYCLE = 2**16 // PERIOD
```

Define constants for the duty_cycle.

- Constants `PERIOD` and `CYCLE` are used by the servo.
- Use the same values as previous missions.

```
12 EXTENDED = 50                 # Percent for servo 45 degrees clockwise
13 RETRACTED = 100               # Percent for servo 45 degrees counterclockwise
```

Define constants for the percents needed to rotate the landing gear.

- 45 degrees clockwise for `EXTENDED`.
- 45 degrees counterclockwise for `RETRACTED`.

*These are the same percents from the chart in **Solar Tracking**.*

```
14
15  # Set up the peripherals
16  power.enable_periph_vcc(True)
17  np = neopixel.NeoPixel(exp.PORT0, 8)
18  ground_sensor = exp.digital_in(exp.PORT1, pull=digitalio.Pull.UP)
19  touchdown_sensor = exp.digital_in(exp.PORT2)
20  landing_gear = exp.pwm_out(exp.PORT3, frequency=PERIOD) # 180 servo
```

Set up the servo to use PWM on PORT3.

- Define the variable as `landing_gear`.
- Use the constant `PERIOD`.

```
21
22  # Set all pixels to one color
23  def set_lighting(rgb_color):
24      for pixel in range(8):
25          np[pixel] = rgb_color
26
27  # Set the servo by calculating the duty_cycle
28  def set_servo(percent):
29      return CYCLE * percent // 100
```

Define the function `set_servo()` to calculate the duty_cycle.

- Use one parameter `percent`.
- This is the same function used in **Mission Life Support** and **Solar Tracking**.

This function will be used to rotate the landing gear.

```
30
31  # Main program
32  # Initialize the first state and let object sensor settle
33  lander_state = 'init'
34  set_lighting(RGB_YELLOW)
35  # Retract landing gear
36  landing_gear.duty_cycle = set_servo(RETRACTED)
```

Initialize the landing gear to be **retracted**.

```
37  sleep(1)
38
39  while True:
40      if lander_state == 'init':
41          # Read object sensor and extend landing gear
42          if ground_sensor.value == GROUND_DETECTED:
43              lander_state = 'prepare'
44              set_lighting(RGB_RED)
45              landing_gear.duty_cycle = set_servo(FORWARD)
```

Extend the landing gear to prepare for landing.

```
46      elif lander_state == 'prepare':
47          # Read microswitch
48          if touchdown_sensor.value == TOUCHDOWN_DETECTED:
49              lander_state = 'landed'
50              set_lighting(RGB_GREEN)
51      elif lander_state == 'landed':
52          break
```

Hint:

- ## Familiar Code

  This mission uses the 180 servo, just like **Project Solar Tracking**.

  - The code for this mission will use much of the same code as the previous mission.
  - You may want to open **Solar Tracking** and review the code.
  - You can even copy and paste some of the code.

Goals:

- Define the constants `PERIOD`, `CYCLE`, `EXTENDED`, and `RETRACTED`.
- Define the function `set_servo(percent)`.
- Assign `landing_gear.duty_cycle` at least twice.
- Call `set_servo()` at least twice.

Tools Found: Servos, State, Constants, Functions

Solution:

```
1  from codex import *
2  from time import sleep
3  
4  # Define constants for the peripherals
5  RGB_YELLOW = (20, 20, 0)
6  RGB_RED = (20, 0, 0)
7  RGB_GREEN = (0, 20, 0)
8  GROUND_DETECTED = True       # Object sensor
9  TOUCHDOWN_DETECTED = False   # Microswitch
10 PERIOD = 20                  # Milliseconds / Hz
11 CYCLE = 2**16 // PERIOD
12 EXTENDED = 50                # Percent for servo 45 degrees clockwise
13 RETRACTED = 100              # Percent for servo 45 degrees counterclockwise
14 
15 # Set up the peripherals
16 power.enable_periph_vcc(True)
17 np = neopixel.NeoPixel(exp.PORT0, 8)
18 ground_sensor = exp.digital_in(exp.PORT1, pull=digitalio.Pull.UP)
19 touchdown_sensor = exp.digital_in(exp.PORT2)
20 landing_gear = exp.pwm_out(exp.PORT3, frequency=PERIOD)
21 
22 # Set all pixels to one color
23 def set_lighting(rgb_color):
24     for pixel in range(8):
25         np[pixel] = rgb_color
26 
27 # Set the servo by calculating the duty_cycle
28 def set_servo(percent):
29     return CYCLE * percent // 100
30 
31 # Main program
32 # Initialize the first state and let object sensor settle
33 lander_state = 'init'
34 set_lighting(RGB_YELLOW)
35 # Retract landing gear
36 landing_gear.duty_cycle = set_servo(RETRACTED)
37 sleep(1)
38 
39 while True:
40     if lander_state == 'init':
41         # Read object sensor and extend landing gear
42         if ground_sensor.value == GROUND_DETECTED:
43             lander_state = 'prepare'
44             set_lighting(RGB_RED)
45             landing_gear.duty_cycle = set_servo(EXTENDED)
46     elif lander_state == 'prepare':
47         # Read microswitch
48         if touchdown_sensor.value == TOUCHDOWN_DETECTED:
49             lander_state = 'landed'
50             set_lighting(RGB_GREEN)
```

```
51      elif lander_state == 'landed':
52          break
```

Mission 8 Complete

You've completed project Prepare Lander!

You can sense nearby objects and determine the state of the landing craft.

The lander is prepared for entry into the atmosphere! The crew is extremely pleased with the new lander's ground sensing system.

Congratulations! You have finished preparations to send the crew to Mars!!

Mission 9 - Automatic Garden!

This project shows you how to connect and use a relay to operate devices like the water pump!

Mission Briefing:

The crew will need to be able to generate their own food when they reach their destination.

- They will require some automation to assist them in production!

Build a system to sense soil moisture levels and automatically water a garden.

Project: Automatic Garden will guide you through automating food production on Mars.

Project Goals:

- Learn about relays
- Add a soil moisture sensor to assist with automation
- Understand small pumps and their operation
- Learn how to prime a pump and what that means
- Automate food production for busy life in space!!

Note:

You will need additional materials for this mission:

- Two cups size 16-oz or larger
- Water
- Some soil
- A small flat-head screwdriver

Objective 1 - Automatic Garden Begin!

Time to build the automated gardening system for the colony on Mars!

Your system will be complete with:

- A soil moisture detector and a water pump to perfectly balance the moisture in the soil.

But....you don't want to stand around and turn the pump on and off. There has to be a better way ... an **automatic** way. You look through your peripherals kit and ... YES ... You can use a **relay** to regulate the pump.

Why would you need to use a relay in your circuit?

A relay is just an electrically operated switch!

- It allows a relatively low power circuit to switch high powered loads.
- It could allow a single controller to switch multiple different loads at the same time.
- It could even allow a DC logic circuit to switch AC Power!

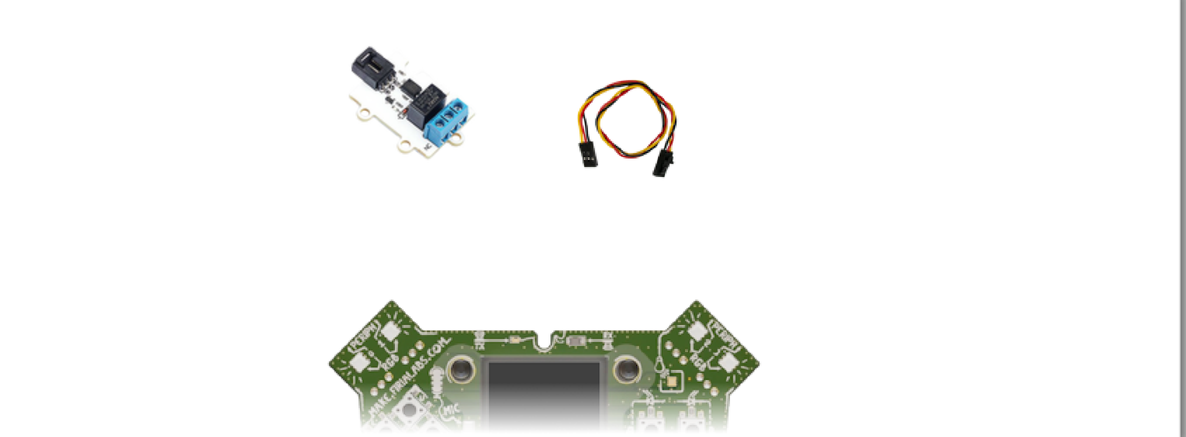
The biggest benefit of a relay is isolation!

- The DC logic circuit can be completely isolated from the load circuit
- The relay will be used to provide external power to the water pump.

Connect Peripheral

Connect the **3V relay** to **PORT0** on your CodeX.

Disconnect all other peripherals from the CodeX.



Create a New File!

Use the **File** → **New File** menu to create a new file called ***AutomaticGarden***.

Check the 'Trek!

Follow the CodeTrek to set up the relay.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Set up peripherals
5  power.enable_periph_vcc(True)
```

The relay requires a lot of power, so you will manually turn the power on.

- Remember to import the libraries you will need.

```
6  relay = exp.digital_out(exp.PORT0)
```

Set up the relay on PORT0.

- Tell the CodeX a digital output peripheral is connected on PORT0.

Hint:

- ## Relay Example

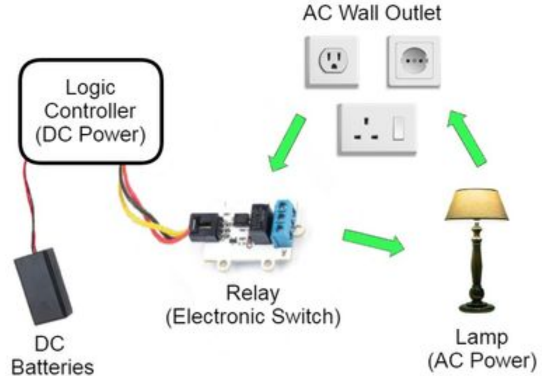
  How about an example of using a relay?

Say you were trying to turn on a bedroom lamp using a battery operated device.

- The lamp requires AC (Alternating Current) power from a wall outlet.
- The battery logic controller would be completely destroyed if you hooked it into AC power.

Both circuits need complete isolation from each other!

Take a look at the diagram to visualize the lamp example.



Goals:

- Create a new file named `AutomaticGarden`.
- Call `power.enable_periph_vcc(True)`.
- Set up the **3V relay** by assigning the variable `relay` as the output of `exp.digital_out(exp.PORT0)`.

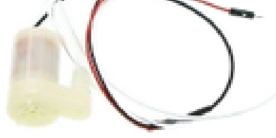
Tools Found: Variables

Solution:

```
1 from codex import *
2 from time import sleep
3
4 # Set up peripherals
5 power.enable_periph_vcc(True)
6 relay = exp.digital_out(exp.PORT0)
```

Objective 2 - Connect the Water Pump!

Time to connect the water pump to the relay.



Mars doesn't have any wall outlets, so the water pump needs an external power source.

- A **two-battery pack** should work!
- But ... you don't have one.
- The CodeX has power, either through the USB or its battery pack.
- Use the CodeX as an external power source.

Check the Hints for more details about power sources.

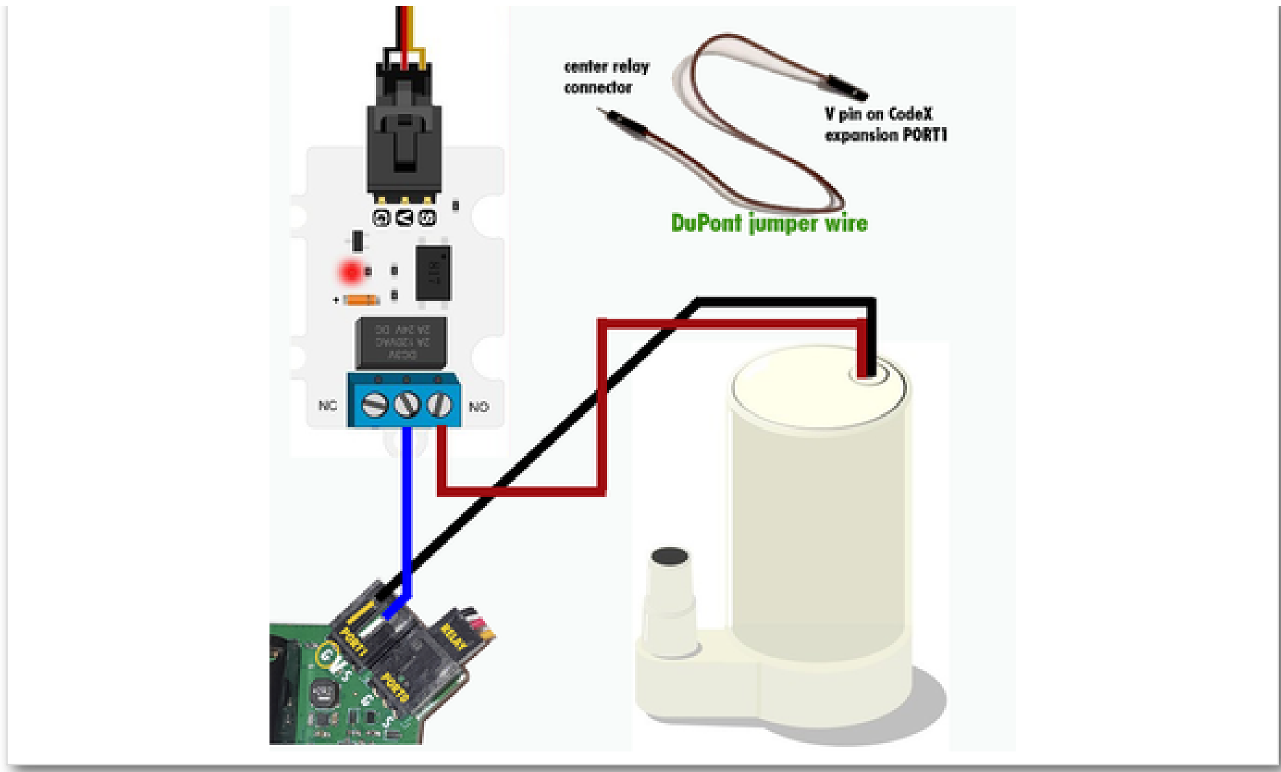
Physical Interaction: *Water Pump*

Connect the **water pump** to the **3V relay** and a **CodeX** as shown in the image below.

- You will use **PORT0** and **PORT1** on your CodeX.
- Connect the **3V relay** to **PORT0** on your CodeX - use a 3-wire connector.
- Connect the **RED wire** from the pump to the **3V relay NO** terminal.
- Connect the **BLACK wire** from the pump to the **PORT1 GND pin** on your CodeX.
- Using a DuPont jumper wire from the kit, connect the **PORT1 V (middle) pin** on your CodeX to the **3V relay Center** terminal.

NOTES:

- You will need a small flat-head screwdriver to loosen and tighten the relay terminals.
- Do not tighten the terminals to the point where the ends of the wire break or bend.
- Tighten just enough to maintain a positive electrical connection!



The relay has three different screw terminals:

- NC (Normally Closed)
- Power Input
- NO (Normally Open)

When connecting the relay you should:

- Always connect the power that you are switching to the **center** screw terminal.
- For this mission, the CodeX V pin will be connected to the **center** screw terminal.
- Use the **NO** screw terminal for any other connection because it will close the circuit when the relay is `True`.
- For this mission, the pump's red wire will be connected to the **NO** screw terminal.

Check the Hints for more details about the relay terminals.

Check the 'Trek!

Follow the CodeTrek to turn on and off the pump.

Run It!

Run the code.

- Does the pump turn on for 2 seconds and then turn off?

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Set up peripherals
5  power.enable_periph_vcc(True)
6  relay = exp.digital_out(exp.PORT0)
7
8  relay.value = True
9  sleep(2)
```

Turn on the pump by setting the relay value property to `True`.

- You are using the **NO** terminal, so the relay's `True` property closes the circuit.
- Delay for 2 seconds.

```
10
11  relay.value = False
12  sleep(2)
```

Turn off the pump by setting the relay value property to `False`.

Hints:

- ## Power Source FAQs

  Do I really need the relay?

  - Yes, to ensure proper operation you will need a relay for the water pump.

  How much power does the pump need?

  - The optimal power supply for the pump would be 3 AA or 3 AAA Batteries.
  - The pump normal operating voltage is 3 - 4.5 Volts DC.

  A single AA or AAA battery produces:

  - ~ 1.5 V when it is new
  - As low as 1.35 V when it is used

- ## Relay terminals

  When connecting the relay you should:

  - Always connect the power that you are switching to the center screw terminal. The other two terminals (NC and NO) are optional:
  - NO (Normally Open) = current will flow from the center screw terminal into the NO terminal **only** when the logic input to the relay is `True` (HIGH).
  - NC (Normally Closed) = current will flow from the center screw terminal into the NC terminal **only** when the logic input to the relay is `False` (LOW).

  NO is the most common usage of a relay.

  - This is because it is more common to want the peripheral to be powered with a HIGH signal and off with a LOW signal.
  - Many relays only offer the NO terminal!

  And yes... you can connect to both terminals at the same time if you want to!

Goals:

- Import `sleep` from `time`.
- Assign `relay.value` at least twice.
- Call `sleep(2)` at least twice.

Tools Found: import

Solution:

```
1  from codex import *
2  from time import sleep
```

```
3
4  # Set up peripherals
5  power.enable_periph_vcc(True)
6  relay = exp.digital_out(exp.PORT0)
7
8  relay.value = True
9  sleep(2)
10
11 relay.value = False
12 sleep(2)
```

Objective 3 **- Pump it Up!**

Add code to operate the water pump!

Many pumps must be primed before operation. Priming is the process of removing air from the pump lines.

- In other words, you need to fill the lines with water before the pump can operate properly.

The first thing you will need is your water source.

- A cup or container of water is perfect.

You will also need a place to pump water to.

- A second empty cup will work!

The next step is to submerge the pump in the water as shown here:

- It's OK! The pump is designed to be submersible.
- Try to keep the bottom of the pump above the bottom of the outflow line.

Now it is time to prime! The easiest way to prime the water pump is:

- Get the pump running and then squeeze water through the line!
- If necessary, you can use your fingers to massage the water through the line.

This looks like a good place to write a function.

Check the 'Trek!

Follow the CodeTrek to define a function for priming the pump.

Physical Interaction: *Try it!*

Press **BTN_A** on the **CodeX** to prime the pump.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  PUMP_ON = True
6  PUMP_OFF = False
```

Define constants for the relay to use: `PUMP_ON` and `PUMP_OFF`.

- Just like you did for other peripherals, defining constants will give meaning to your code.

```
7
8  # Set up peripherals
9  power.enable_periph_vcc(True)
10 relay = exp.digital_out(exp.PORT0)
11
12 def prime_pump():
13     relay.value = PUMP_ON
14     sleep(5)
15     relay.value = PUMP_OFF
```

Define the function `prime_pump()` that will turn the pump on for a period of time.

- Set the relay value property to `PUMP_ON`.
- Then delay for a set amount of time.
- Then turn the pump off - it is primed!

```
16
17 while True:
18     if buttons.was_pressed(BTN_A):
19         prime_pump()
```

Use `buttons.was_pressed()` to call `prime_pump()`.

- The pump may need to be primed on occasion. Using a button press lets you prime the pump on demand.
- Use a `while True:` infinite loop.

Hint:

- ## The Importance of Priming

  The majority of pump problems occur because the pump was not primed first.

  - Gases tend to expand when pushed and block the line.
  - The pump will not generate enough pressure to push water through the gas.
  - Some pumps can even overheat if they are not properly primed!

Goals:

- Define the 🔧constants `PUMP_ON` and `PUMP_OFF`.
- Define the 🔧function `prime_pump()`.
- Call `prime_pump()`.
- Use an `if` statement with the 🔧condition `buttons.was_pressed(BTN_A)`.

Tools Found: Functions, Constants, bool

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  PUMP_ON = True
6  PUMP_OFF = False
7
8  # Set up peripherals
9  power.enable_periph_vcc(True)
10 relay = exp.digital_out(exp.PORT0)
11
12 def prime_pump():
13     relay.value = PUMP_ON
14     sleep(5)
15     relay.value = PUMP_OFF
16
17 while True:
18     if buttons.was_pressed(BTN_A):
19         prime_pump()
```

Objective 4 **- Detect Soil Moisture**

It is time to detect the moisture in the soil!

The soil moisture sensor is an 🔧analog **input** sensor.

It measures the conductivity of the soil between the two sensor tines.

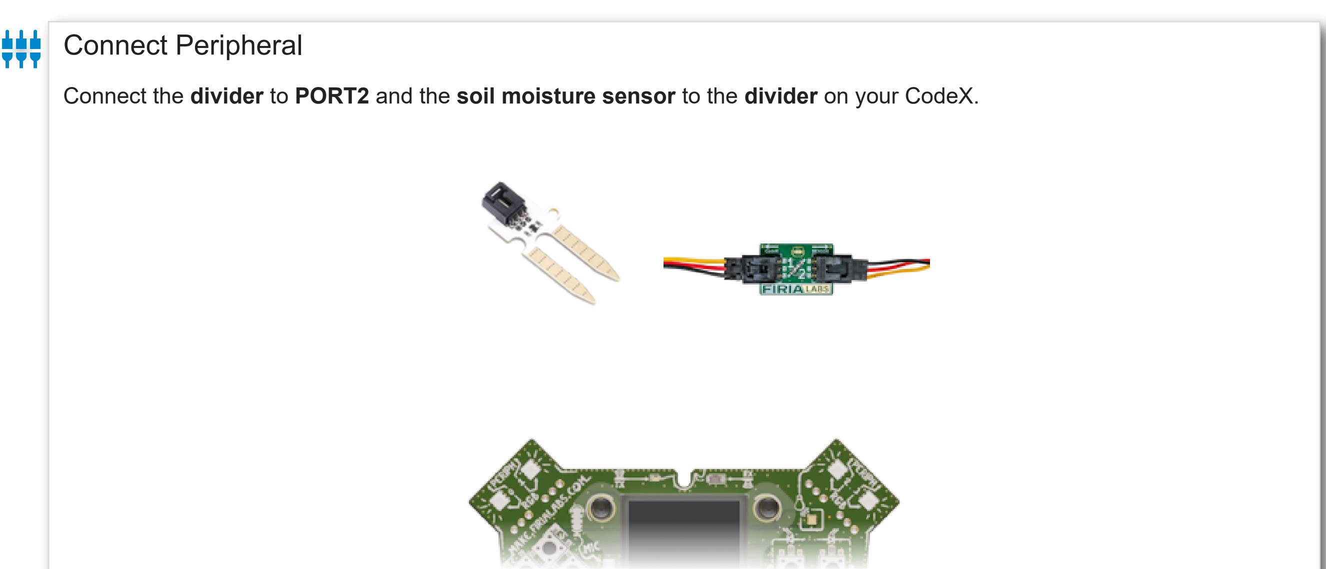
- One tine outputs a very small DC current.
- The second tine reads the current as an input.
- It will return a value between 0-65535.
- Like other analog input devices, the soil moisture sensor will use the resistor voltage divider to limit the ADC input.

More water in the soil = More conductivity = Less resistance

- This means as **moisture increases** the input values will also **increase**!

Connect Peripheral

Connect the **divider** to **PORT2** and the **soil moisture sensor** to the **divider** on your CodeX.



Check the 'Trek!

Follow the CodeTrek to read the soil moisture sensor and display the output on the **Console Panel**.

Physical Interaction: *Try it!*

Place the soil moisture sensor into some soil, and place the end of the water line in the cup with soil.

Run the code. Prime the pump if needed.

- Open the **Console Panel** and observe the soil moisture values.
- You should see a new soil moisture sensor value every 5 seconds.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  PUMP_ON = True
6  PUMP_OFF = False
7  TIME_ON = 1        # Seconds
8  TIME_OFF = 4       # Seconds
```

Define constants for the pump timer: `TIME_ON` and `TIME_OFF`.

- Decide how long you want the pump to stay on (in seconds),
- and how long it should stay off before turning on again.
- The values are small for testing purposes.

```
9
10 # Set up peripherals
11 power.enable_periph_vcc(True)
12 relay = exp.digital_out(exp.PORT0)
13 soil_moisture = exp.analog_in(exp.PORT2)
```

Set up the soil moisture sensor on PORT2.

- Tell the CodeX an analog input peripheral is connected on PORT2.

```
14
15 def prime_pump():
16     relay.value = PUMP_ON
17     sleep(5)
18     relay.value = PUMP_OFF
19
20 while True:
21     if buttons.was_pressed(BTN_A):
22         prime_pump()
23
24     print("Moisture Val: ", soil_moisture.value)
```

Read the moisture sensor and display the output on the **Console Panel**.

- Open the **Console Panel** when you run the program.

```
25
26     relay.value = PUMP_ON
27     sleep(TIME_ON)
28     relay.value = PUMP_OFF
29     sleep(TIME_OFF)
```

While testing the soil moisture sensor, turn on and off the
pump automatically.

- Use the constants defined for the pump timer.

Hint:

- ## Time ON and Time OFF

You can define constants for the pump timing.
- Constants give your code meaning.
- It also makes it easier to change the values.

During this mission, you will define constants for how much time the pump is **ON** and **OFF**.

- You can easily adjust these values for times that work for you and your soil.

Goals:

- Define the 🔧constants `TIME_ON` and `TIME_OFF`.
- Set up the **soil moisture sensor** by assigning the 🔧variable `soil_moisture` as the output of `exp.analog_in(exp.PORT2)`.
- 🔧Print `soil_moisture.value`.
- Call `sleep(TIME_ON)` and `sleep(TIME_OFF)`.

Tools Found: Analog to Digital Conversion, Constants, Variables, Print Function

Solution:

```
1  from codex import *
2  from time import sleep
3  
4  # Define constants for the peripherals
5  PUMP_ON = True
6  PUMP_OFF = False
7  TIME_ON = 1         # Seconds
8  TIME_OFF = 4        # Seconds
9  
10 # Set up peripherals
11 power.enable_periph_vcc(True)
12 relay = exp.digital_out(exp.PORT0)
13 soil_moisture = exp.analog_in(exp.PORT2)
14 
15 def prime_pump():
16     relay.value = PUMP_ON
17     sleep(5)
18     relay.value = PUMP_OFF
19 
20 while True:
21     if buttons.was_pressed(BTN_A):
22         prime_pump()
23 
24     print("Moisture Val: ", soil_moisture.value)
25 
26     relay.value = PUMP_ON
27     sleep(TIME_ON)
28     relay.value = PUMP_OFF
29     sleep(TIME_OFF)
```

Quiz 1 - Check Your Understanding: Relays and Moisture Sensors

Question 1: What is the biggest benefit of using a relay?

✓ Circuit isolation

✗ An extra switch

✗ More available power

✗ An extra port for peripherals

Question 2: What type of peripheral is a **relay**?

✓ Digital output

❌ Digital input

❌ Analog output

❌ Analog input

Question 3: What **relay** screw terminal is the water pump connected to?

❌ Center

❌ NC

✅ NO

❌ Any of them will work

Question 4: What type of peripheral is the soil moisture sensor?

✅ Analog input

❌ Analog output

❌ Digital input

❌ Digital output

Question 5: What values are returned by a moisture sensor reading?

✅ Integers between 0-65535

❌ `True` or `False`

❌ Integers between 0-10

❌ `PUMP_ON` or `PUMP_OFF`

Objective 5 - Fine Tune Automation

Add a Low Moisture Threshold to Start the Pump Timer

For testing, you set the pump to run an on/off cycle. But the garden may not always need constant watering.

Add a `LOW_MOISTURE_THRESHOLD` constant to your code.

- This will be used to trigger the pump **ON** for a `TIME_ON` delay.
- Look at the moisture sensor readings on the **Console Panel**.
- Decide on a value that would mean dry soil that needs water.
- During my testing, dry soil was at a reading below `10000`.

Then add an `if` condition in the `while` loop to check the reading against your new threshold.

Now you just need to fine tune your constants a little!

During testing, `TIME_ON` and `TIME_OFF` were given small values.

- Decide how long the pump should run to nourish the garden.
- This is still a simulation, so the number will remain small but can be more than 1 second.
- Decide how long you want the water to stay off in between readings.
- This will give the soil a little time to absorb water before taking another reading!
- You can adjust both constant values as needed.

Check the 'Trek!

Follow the CodeTrek to fine tune the automatic watering system.

Physical Interaction: *Try it!*

Run the code.

- Test that the threshold is working by pulling the sensor out of the soil for a reading.
- This will create a high resistance through the air and return a value well below `LOW_MOISTURE_THRESHOLD`.

CodeTrek:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  PUMP_ON = True
6  PUMP_OFF = False
7  TIME_ON = 1                    # Seconds
8  TIME_OFF = 4                   # Seconds
9  LOW_MOISTURE_THRESHOLD = 10000 # Soil moisture threshold for dry soil
```

Define a constant for dry soil.

- `LOW_MOISTURE_THRESHOLD` will be a value below the normal sensor reading.

```
10
11 # Set up peripherals
12 power.enable_periph_vcc(True)
13 relay = exp.digital_out(exp.PORT0)
14 soil_moisture = exp.analog_in(exp.PORT2)
15
16 def prime_pump():
17     relay.value = PUMP_ON
18     sleep(5)
19     relay.value = PUMP_OFF
20
21 while True:
22     if buttons.was_pressed(BTN_A):
23         prime_pump()
24
25     print("Moisture Val: ", soil_moisture.value)
26
27     if soil_moisture.value < LOW_MOISTURE_THRESHOLD:
```

Compare the current moisture sensor reading to `LOW_MOISTURE_THRESHOLD`.

- If the soil is dry, the condition is `True` and it will trigger the pump to turn **ON**.

```
28         relay.value = PUMP_ON
29         sleep(TIME_ON)
30         relay.value = PUMP_OFF
```

When the condition is `True`:

- Turn the pump **ON**.
- Use the `TIME_ON` delay.
- Then turn the pump **OFF**.

```
31
32     sleep(TIME_OFF)
```

Delay for awhile before reading the soil moisture sensor again.

- This gives the soil time to absorb the water.

- Also, you don't want to check the soil too frequently.
- **NOTE:** This delay is **NOT** part of the `if` statement. Watch your indenting!

Hint:

- ## How much water?

  You need to determine the appropriate soil moisture level for your plant.

  All plants are different!

  You can adjust the moisture level by tweaking the LOW_MOISTURE_THRESHOLD.

  - Try some different values and see what your plant needs!
  - You can also adjust the `TIME_ON` and `TIME_OFF` constants.

Goals:

- Define the constant `LOW_MOISTURE_THRESHOLD`.
- Use an `if` statment with the condition `soil_moisture.value < LOW_MOISTURE_THRESHOLD`.
- In the `if` statement from **goal 2**:
- Assign `relay.value`.
- Call `sleep(TIME_ON)`.

Tools Found: Constants, bool

Solution:

```
1  from codex import *
2  from time import sleep
3
4  # Define constants for the peripherals
5  PUMP_ON = True
6  PUMP_OFF = False
7  TIME_ON = 1                       # Seconds
8  TIME_OFF = 4                      # Seconds
9  LOW_MOISTURE_THRESHOLD = 10000  # Soil moisture threshold for dry soil
10
11 # Set up peripherals
12 power.enable_periph_vcc(True)
13 relay = exp.digital_out(exp.PORT0)
14 soil_moisture = exp.analog_in(exp.PORT2)
15
16 def prime_pump():
17     relay.value = PUMP_ON
18     sleep(5)
19     relay.value = PUMP_OFF
20
21 while True:
22     if buttons.was_pressed(BTN_A):
23         prime_pump()
24
25     print("Moisture Val: ", soil_moisture.value)
26
27     if soil_moisture.value < LOW_MOISTURE_THRESHOLD:
28         relay.value = PUMP_ON
29         sleep(TIME_ON)
30         relay.value = PUMP_OFF
31
32     sleep(TIME_OFF)
```

Mission 9 Complete

You've completed project Automatic Garden!

You have detected soil moisture and incorporated a water pump.

The tools that you created will be the core of the fully automated food production centers on Mars!

The crew will be able to grow their own food as they begin life outside of Earth!!

Mission 10 - Exploring the Surface!

This project introduces external peripherals that can be added to the CodeX using a breadboard!

Mission Briefing:

Now that your Mars habitat is set up, it's time to fire up the **Martian Rover** to explore the surface. But all around the landing zone, there are large boulders that could damage the Rover.

Project: Exploring the Surface will guide you through keeping the **Rover** safe by adding a distance sensing system.

You will need to *detect the distance to objects* and alert the crew when the Rover is *approaching* or *very close to* a harmful object.

Project Goals:

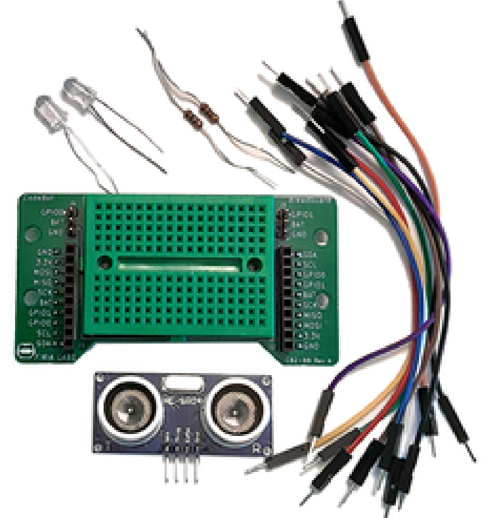
- Connect a **breadboard** to the CodeX
- Connect **external peripherals** to the **breadboard**
- Use a **sonar** to detect distance to an object
- Convert raw data to a distance in centimeters
- Create a warning system for the **Rover**!

Objective 1 - Hard Knocks

Hard Knocks - and a Discovery!

Upon driving the Rover out of the landing zone, it immediately crashed into a boulder which was very difficult to see visually, blending in to the sandy surface texture. The damage was easy to fix this time ... but scary!

While inspecting the Rover, the astronauts discovered the remains of a long-lost spacecraft which crashed on the surface a decade ago! A few parts could be salvaged:



- a breadboard
- a red and an amber LED
- two 100 Ohm resistors
- an HC-SR04 ultrasonic distance sensor
- 10 jumper wires

You decide to use the spare parts to craft a **distance detection warning system** for the Rover. You can:

- use the ultrasonic distance sensor to detect the distance to an object,
- use the amber LED as a warning indicator when the Rover is *approaching* an object,
- and use the red LED as an alert indicator when the Rover is *very close* to a harmful object.

All the parts can be connected on the breadboard, which is then mounted on the Rover.

Check the Hints to learn about breadboards.

Physical Interaction: *Breadboard*

Disconnect all peripherals from the CodeX.

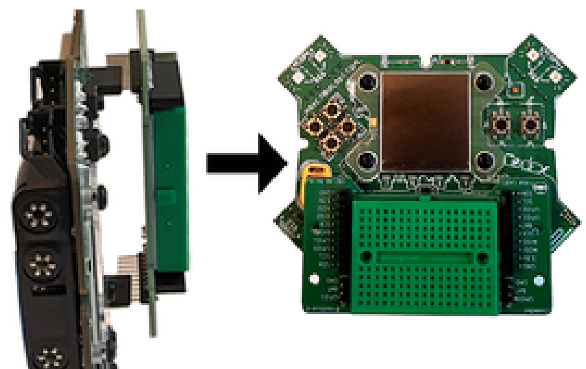
Connect the **breadboard** to **the expansion ports** on your CodeX.

- The **expansion ports** are located on the front of the CodeX, below the buttons.
- You do not want to cover up the display screen, so turn the **breadboard** *upside down*.

The two **expansion ports** on the CodeX have five pin slots each.

- The breadboard has 10 pins on each side.

- Use only five pins on each side of the breadboard.
- The other five pins on each side will not connect to anything.
- Line up the pins with the slots and gently press until firmly in place.



Create a New File!

Use the **File → New File** menu to create a new file called ***ExploreSurface***.

Hints:

- ## Breadboard?

  The term "breadboard" comes from the early days of electronics, when people would literally drive nails or screws into wooden boards on which they cut bread in order to build circuits.

  Today a breadboard is a rectangular plastic board with a bunch of tiny holes in it. The breadboard for this project is a mini breadboard, one of the smallest breadboards available.

- ## A Breadboard and Peripherals

  The tiny holes on the breadboard let you easily insert electronic components to build a circuit.

  - The components can be external peripherals like LEDs, sensors, buttons, and speakers.
  - The connections are not permanent, so it is easy to remove components and reuse the breadboard for a different project.
  - The CodeX will supply power to the mounted breadboard, which can then power the added components.

- ## Jumper Wires

  Jumper wires are used to make connections on a breadboard. Their ends are easy to push into the breadboard holes. The wires come in varying colors. It doesn't matter which colors you use, but color coding the project helps you stay organized and keep track of what you are connecting.

Goal:

- After connecting the breadboard, create a new file named `ExploreSurface`.

Solution:

N/A

***Objective 2* - Going the Distance!**

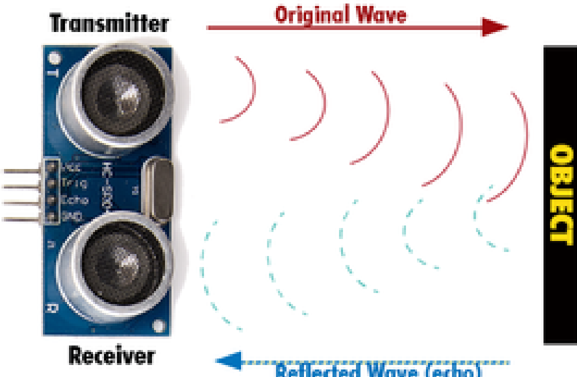
Add the Ultrasonic Sensor.

Getting the HC-SR04 ultrasonic distance sensor is a great find! It can ***detect*** if there is an object ***and*** measure the ***distance to*** the object.

Concept: *Ultrasonic Sensor*

The ultrasonic sensor uses sonar to determine the distance to an object. It follows these steps:

1. The ultrasound transmitter (trig pin) emits a high-frequency sound (40 kHz).
2. The sound travels through the air.
3. If an object is in its path, the sound bounces back.
4. The ultrasound receiver (echo pin) receives the reflected sound (echo).



The sensor then returns the time it took to give and receive the signal. The time between the transmission and reception of the signal allows us to calculate the distance to an object. This is possible because we know the sound's velocity in the air.

- You will use the **Distance=Rate*Time** formula to calculate the distance.

Does the ultrasonic sensor work on Mars? Yes, but mind the speed of sound! For now, test the device using the speed of sound on Earth.

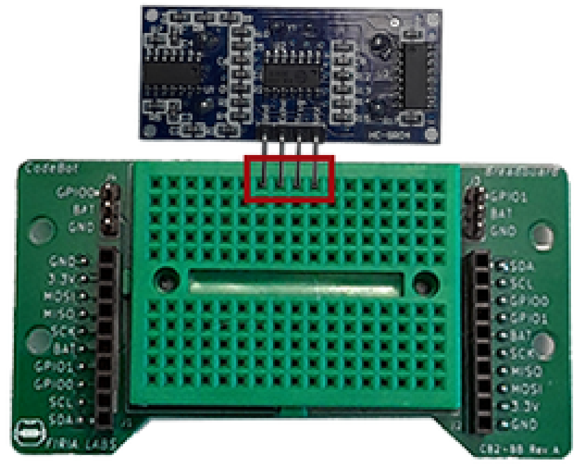
Check the Hints to see the specs of the ultrasonic sensor.

Let's first get the ultrasonic sensor connected and see what the raw data looks like.

Physical Interaction: *Ultrasonic Sensor*

Insert the **ultrasonic sensor** into the breadboard.

- In order to keep a clear path in front of the sensor, use the center holes on the front row of the breadboard.
- The sensor will face *away* from the CodeX and breadboard.



Now the peripheral needs power, a ground, and input/output connections.

Connect the terminals of the ultrasonic sensor to connection points on the breadboard.

- Check the Hints for more information about breadboards and jumper wires.

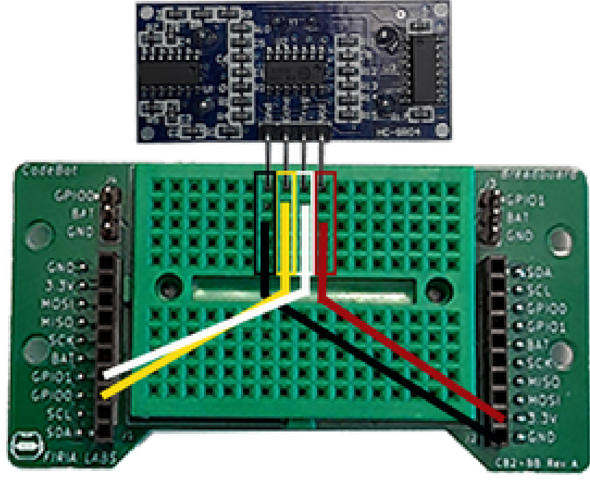
Physical Interaction: *Jumper Wires*

When connecting the peripheral to the breadboard, only use the connection points that are connected to the CodeX.

Connect:

- Sensor's GND pin to the breadboard's GND pin hole
- Sensor's VCC pin to the breadboard's 3.3V pin hole
- Sensor's Echo pin to the breadboard's GPIO0 pin hole
- Sensor's Trig pin to the breadboard's GPIO1 pin hole

NOTE: *You can use any pin hole in the terminal strip that the sensor's pin is in.*



Write some code to get data from the ultrasonic sensor.

- You will need to import a new library.
- The trigger is a digital output device, but the echo will receive a pulse (input).

```
import pulseio
# Set up peripherals
echo = pulseio.PulseIn(exp.GPIO0)
trigger = exp.digital_out(exp.GPIO1)
```

Check the 'Trek!

Follow the CodeTrek to:

- import modules
- set up the peripherals
- write a function for reading data from the sensor
- use the function in the main program to display the data

That is a lot of code!

Time to give it a try.

Run It!

- Open the **console panel** and run the code.
- Put an object in front of the sensor.
- Move the object closer and farther away from the sensor.
- Observe the values printed for each distance.

CodeTrek:

```
1  from codex import *
2  import time
3  import pulseio
```

Import the modules (libraries) you need.

- This includes the new library `pulseio`.

```
4
5  # Set up peripherals
6  echo = pulseio.PulseIn(exp.GPIO0)
7  trigger = exp.digital_out(exp.GPIO1)
```

Set up the trigger and echo as peripherals.

- The **echo** is a pulse input device.
- The **trigger** is a digital output device.

```
8
9  # Use sonar to find the distance to an object
10 def object_distance():
```

Create a function for the ultrasonic sensor.

- It will use sonar to return the distance to an object.

```
11     # Turn on and off signal, clear the echo
12     trigger.value = True
13     trigger.value = False
14     echo.clear()
```

Start the function by turning **on** and **off** the trigger.

- This will transmit a high-frequency sound.
- You also need to clear the echo pin so it is ready to receive the newly transmitted signal.

```
15     # Wait for the echo signal
16     while not echo:
17         pass
18     return echo[0]
```

Use a `while not echo:` loop to wait for the echo signal.

- The loop will not do anything while waiting.
- When an echo is received, return the value to the function call.

```
19
20 # Main program
21 while True:
```

Start the main program with a `while True:` loop.

- You will continuously read the sensor in the loop.

```
22      echo_signal = object_distance()
23      print(echo_signal)
24      time.sleep(2)
```

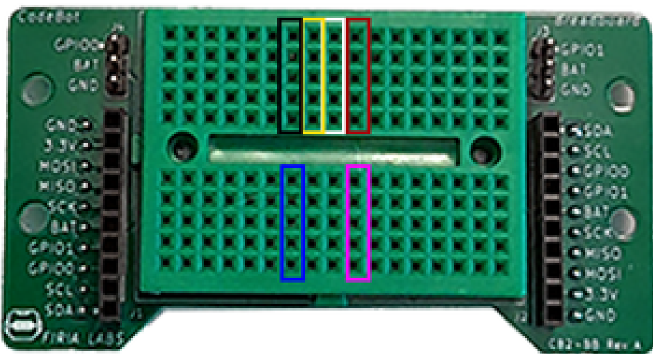
In the main program:

- Call the sensor's function and assign the returned echo value to a variable.
- Display the value in the **console log**.
- Use a short 🔧delay so the data display is not overwhelming.

Hints:

- ## HC-SR04 Ultrasonic Sensor
  - This sensor reads from 2 cm to 400 cm (0.8 inch to 157 inch)
  - It has an accuracy of 0.3cm (0.1 inches)
  - Resolution: 0.3 cm
  - Measuring Angle: 30 degrees
- ## Breadboards

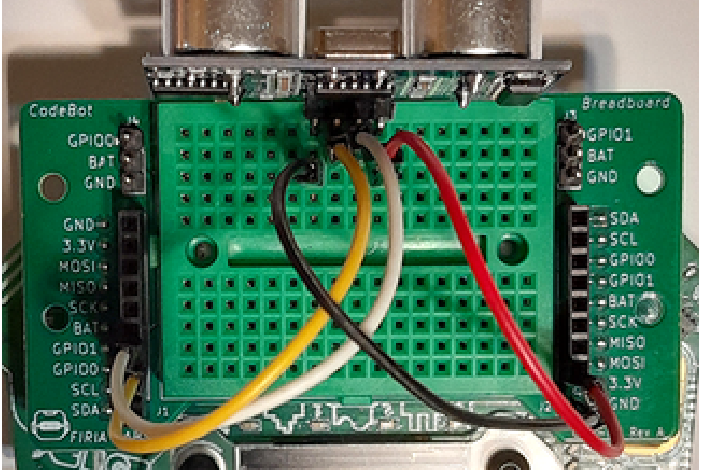
  A breadboard holds your parts steady and has internal wiring to make connections super fast.
  - Each column of holes is a terminal strip.
  - Any parts plugged into the same terminal strip are electrically connected together.
  - The terminal strips on either side of the center divider are ***not*** connected together.



- ## Jumper Wires

  Use jumper wires to electronically connect the terminals of a peripheral to power, ground, or input/output connection points on the breadboard.
  - The color of the jumper wire does not matter.
  - However, color coding your project helps you stay organized.
  - One end of the jumper wire plugs into a hole in the terminal strip.
  - The other end of the jumper wire plugs into a labeled connection point on the side of the breadboard.
- ## Breadboard Setup

  The setup of the ultrasonic sensor may look like this:



Goals:

- Set up the **echo receiver** by assigning the 🔧variable `echo` as the output of `pulseio.PulseIn(exp.GPIO0)`.
- Set up the **trigger transmitter** by assigning the variable `trigger` as the output of `exp.digital_out(exp.GPIO1)`.
- Define the 🔧function `object_distance()`.
- At the end of `object_distance()`, `return echo[0]`.
- Assign `echo_signal` the return value of `object_distance()`.
- 🔧Print `echo_signal` in the **console panel**.

Tools Found: Functions, Variables, Print Function, Timing

Solution:

```
1  from codex import *
2  import time
3  import pulseio
4
5  # Set up peripherals
6  echo = pulseio.PulseIn(exp.GPIO0)
7  trigger = exp.digital_out(exp.GPIO1)
8
9  # Use sonar to find the distance to an object
10 def object_distance():
11     # Turn on and off signal, clear the echo
12     trigger.value = True
13     trigger.value = False
14     echo.clear()
15     # Wait for the echo signal
16     while not echo:
17         pass
18     return echo[0]
19
20 # Main program
21 while True:
22     echo_signal = object_distance()
23     print(echo_signal)
24     time.sleep(2)
```

Objective 3 - Convert to Centimeters

Look at the raw data.

The ultrasonic sensor returns the time between the transmission and reception of the signal in microseconds. The time itself isn't that useful, but you can use it to calculate the distance from the sensor to the object.

Concept: *Converting time to distance*

The basic formula **distance = rate * time** can be used to convert the echo signal (raw data time) to a distance measurement. Since the test Rover CodeX is pretty small you will convert to centimeters.

When using the conversion formula:

- Use **half** the time; you only want the *return* time, not the total time.
- The **rate** is the speed of sound.
- The speed of sound will be different on Mars than it is on Earth, so use a constant for this value. Then it will be easy to change later.

Check the Hints to learn more about the speed of sound.

Add a function to convert the echo signal's raw time to centimeters.

Run It!

CodeTrek:

```
1  from codex import *
2  import time
3  import pulseio
4
5  # Constants for peripherals
6  SOUND_SPEED = 34300      # Rate: centimeters / second
```

Define a constant SOUND_SPEED for the rate.

- This is the speed of sound on Earth.
- The speed is 343 m/sec, or 34300 cm/sec.
- The value can be changed for the speed of sound on Mars.

```
7
8  # Set up peripherals
9  echo = pulseio.PulseIn(exp.GPIO0)
10 trigger = exp.digital_out(exp.GPIO1)
11
12 # Convert echo time from microseconds to seconds
13 def convert_to_centimeters(echo_signal):
```

Define a function `convert_to_centimeters()` that will take the `echo_signal` returned by the ultrasonic sensor as a parameter.

- Then use the formula: **distance = rate * time**

```
14     echo_time = echo_signal / 1000000
```

First convert the `echo_signal` parameter from microseconds to seconds.

```
15     # Calculate using distance = rate * time
16     return SOUND_SPEED * (echo_time / 2)
```

Use the **distance = rate * time** formula to calculate the distance in centimeters.

- Use **half** the time; you only need the return time.
- Return the **distance in centimeters** to the function call.

```
17
18 # Use sonar to find the distance to an object
19 def object_distance():
```

```
20      # Turn on and off signal, clear the echo
21      trigger.value = True
22      trigger.value = False
23      echo.clear()
24      # Wait for the echo signal
25      while not echo:
26          pass
27      # convert raw data (time) to centimeters
28      distance_cm = convert_to_centimeters(echo[0])
```

Assign a value to `distance_cm` in the `object_distance()` function by calling the `convert_to_centimeters()` function.

- Use the ultrasonic sensor's return value `echo[0]` as the parameter.

```
29      return distance_cm
```

Return the distance `distance_cm` to the main program.

```
30
31  # Main program
32  while True:
33      distance_cm = object_distance()
34      print(distance_cm, "centimeters")
```

Modify the main program to:

- Assign a value to `distance_cm` instead of `echo_signal`.
- Print `distance_cm` instead of `echo_signal`.

```
35      time.sleep(2)
```

Hints:

- ## Speed of Sound (on Earth)

  The speed of sound in the air at 20 degrees C (68 degrees F) = 343m/s

  - You want to convert to centimeters, so multiply 343 * 100

    1 meter = 100 centimeters

  - The time returned by the sensor is in microseconds.
  - You will need to divide the time by 1000000 to convert it to seconds.

    1 second = 1000000 microseconds

- ## Sounds on Mars

  On Mars, the atmosphere is entirely different than on Earth. Sounds on Mars will be a little quieter and more muffled than what you hear on Earth. Also, you'll wait slightly longer to hear sounds after they are transmitted.

  But, the biggest change to audio will be high-pitch sounds. Some sounds, like whistles, bells or bird songs, will almost be inaudible on Mars.

Goals:

- Define a 🔧constant named `SOUND_SPEED`.
- Define the 🔧function `convert_to_centimeters(echo_signal)`.
- Assign the 🔧variable `distance_cm` as the return of `convert_to_centimeters()`.
- 🔧Print `distance_cm` in the **console panel**.

Tools Found: Constants, Functions, Print Function, Variables

Solution:

```
1  from codex import *
2  import time
3  import pulseio
4  
5  # Constants for peripherals
6  SOUND_SPEED = 34300     # Rate: centimeters / second
7  
8  # Set up peripherals
9  echo = pulseio.PulseIn(exp.GPIO0)
10 trigger = exp.digital_out(exp.GPIO1)
11 
12 # Convert echo time from microseconds to seconds
13 def convert_to_centimeters(echo_signal):
14     echo_time = echo_signal / 1000000
15     # Calculate using distance = rate * time
16     return SOUND_SPEED * (echo_time / 2)
17 
18 # Use sonar to find the distance to an object
19 def object_distance():
20     # Turn on and off signal, clear the echo
21     trigger.value = True
22     trigger.value = False
23     echo.clear()
24     # Wait for the echo signal
25     while not echo:
26         pass
27     # convert raw data (time) to centimeters
28     distance_cm = convert_to_centimeters(echo[0])
29     return distance_cm
30 
31 # Main program
32 while True:
33     distance_cm = object_distance()
34     print(distance_cm, "centimeters")
35     time.sleep(2)
```

Objective 4 - Time Out!

What if an object isn't close by?

The ultrasonic sensor waits to receive a return signal. But what if there isn't an object to bounce the signal off of? A return signal won't be sent.

The `object_distance()` function has a loop that waits for a return signal, so if one isn't sent, that could be a problem.

You reflect on the programming techniques you used earlier during these missions. And then it comes to you!

- Include a **time out** feature in the loop so that it won't keep waiting.
- Use the CodeX's internal **Timer** to keep track of the time the loop waits.
  - From the moment you connect the CodeX, the **Timer** is *always running*.
- If the loop waits too long, break the loop and return a value that indicates no object is close by.

Type in the Code

Define a constant for the wait time:

```
WAIT_TIME = 100        # Milliseconds
```

In the `object_distance()` function, change:

```
    # Wait for the echo signal
    while not echo:
        pass
```

to:

```
    # Return time for echo or time out if no object detected
    echo_start = time.ticks_ms()
    while not echo:
        if time.ticks_ms() > WAIT_TIME + echo_start:
            return -1
```

The `return -1` will break the loop and return the value `-1` for the time.

Wait a minute (or 100 milliseconds)!

A negative return time!? That will convert to a negative distance.

You can use that negative distance in your code to mean "no object detected".

Check the 'Trek!

Follow the CodeTrek to modify the main program by adding a condition for a negative distance.

Test your code again.

Run It!

- Open the **console panel** and run the code.
- Put an object in front of the sensor.
- Point the sensor away from all objects.
- Make sure the correct statement is displayed.

Does the **console log** show "No object detected" when the distance is far away?

CodeTrek:

```
1  from codex import *
2  import time
3  import pulseio
4
5  # Constants for peripherals
6  SOUND_SPEED = 34300      # Rate: centimeters / second
7  WAIT_TIME = 100          # Milliseconds
```

Define a constant `WAIT_TIME` for the loop.

- Use the **milliseconds** time measurement.

```
8
9  # Set up peripherals
10 echo = pulseio.PulseIn(exp.GPIO0)
11 trigger = exp.digital_out(exp.GPIO1)
12
13 # Convert echo time from microseconds to seconds
14 def convert_to_centimeters(echo_signal):
15     echo_time = echo_signal / 1000000
16     # Calculate using distance = rate * time
17     return (echo_time / 2) * SOUND_SPEED
18
19 # Use sonar to find the distance to an object
20 def object_distance():
21     # Turn on and off signal, clear the echo
22     trigger.value = True
23     trigger.value = False
24     echo.clear()
25     # Return time for echo or time out if no object detected
26     echo_start = time.ticks_ms()
```

Define a variable for the starting time.

- Use the current clock time.

```
27      while not echo:
28          if time.ticks_ms() > WAIT_TIME + echo_start:
29              return -1
```

Modify the body of the loop to constantly check if the current time is more than `WAIT_TIME + echo_start`.

- If time is up, return `-1` and break the loop.
- The loop will automatically stop if an object is detected.

```
30      distance_cm = convert_to_centimeters(echo[0])
31      return distance_cm
32
33  # Main program
34  while True:
35      distance_cm = object_distance()
36
37      if distance_cm < 0:
38          print("No object detected")
39      else:
40          print("Object detected:", distance_cm, "centimeters")
```

Add an `if` statement to the main program that checks for a negative distance.

- If **negative**, no object is detected.
- Otherwise, print the distance to the object.

```
41
42      time.sleep_ms(WAIT_TIME)
```

Now that your code has been tested, and the Rover is ready to go, you will want to read the sensor more frequently.

- Change the time from seconds to milliseconds.
- Use `WAIT_TIME` for the delay.

Hints:

- ## The CodeX Timer

  You used the **Timer** in **Project 3: Conserve Energy**. You may want to open the mission and review Objectives 6 & 7.

  For this project, you are using the **Timer** in a similar yet different way.

  Go to the toolbox and learn even more about the 🔧time module.

- **Timer Similarities**
  - A starting time is needed.
  - A pre-determined **WAIT** time was defined.
  - The starting and **WAIT** times are added together for the ending time.
  - The current time is compared to the ending time.

- **Timer Differences**
  - For **Conserve Energy**, the **Timer** was used as a condition for the `while` loop.
  - The loop will continue until the ending time is reached.
  - For **Exploring the Surface**, the **Timer** is used as a condition for an `if` statement.
  - The loop constantly checks the time and stops if the ending time is reached.
  - The loop will also stop when the looping condition `not echo` is `False`.

Goals:

- Define the 🔧constant `WAIT_TIME` in milliseconds.
- Use an `if` statement with the 🔧condition `time.ticks_ms() > WAIT_TIME + echo_start`.
- Use an `if` statement with the condition `distance_cm < 0`.
- Call `time.sleep_ms(WAIT_TIME)`.

Tools Found: Parameters, Arguments, and Returns, Constants, bool, Variables

Solution:

```
1  from codex import *
2  import time
3  import pulseio
4  
5  # Constants for peripherals
6  SOUND_SPEED = 34300     # Rate: centimeters / second
7  WAIT_TIME = 100         # Milliseconds
8  
9  # Set up peripherals
10 echo = pulseio.PulseIn(exp.GPIO0)
11 trigger = exp.digital_out(exp.GPIO1)
12 
13 # Convert echo time from microseconds to seconds
14 def convert_to_centimeters(echo_signal):
15     echo_time = echo_signal / 1000000
16     # Calculate using distance = rate * time
17     return (echo_time / 2) * SOUND_SPEED
18 
19 # Use sonar to find the distance to an object
20 def object_distance():
21     # Turn on and off signal, clear the echo
22     trigger.value = True
23     trigger.value = False
24     echo.clear()
25     # Return time for echo or time out if no object detected
26     echo_start = time.ticks_ms()
27     while not echo:
28         if time.ticks_ms() > WAIT_TIME + echo_start:
29             return -1
30     distance_cm = convert_to_centimeters(echo[0])
31     return distance_cm
32 
33 # Main program
34 while True:
35     distance_cm = object_distance()
36 
37     if distance_cm < 0:
38         print("No object detected")
39     else:
40         print("Object detected:", distance_cm, "centimeters")
41 
42     time.sleep_ms(WAIT_TIME)
```

***Quiz 1* - Check Your Understanding: Sonar Sensor**

Question 1: The **VCC** terminal of the ultrasonic sensor is connected to what connection point on the breadboard?

✔ 3.3V

✘ GND

✘ GPIO0

✘ GPIO1

Question 2: What data is transmitted by the ultrasonic sensor?

✓ High-frequency sound

✗ Transmission time

✗ Distance to the object

✗ Raw time data

Question 3: What information is returned by the ultrasonic sensor?

✓ Transmission and receiving time in microseconds

✗ Total time in milliseconds

✗ Distance to the object

✗ The sound wave

Question 4: What is the formula for finding the distance to an object when using a sonar reading?

✓ distance = SOUND_SPEED * echo_time / 2

✗ distance = SOUND_SPEED * echo_time

✗ distance = SOUND_SPEED / echo_time

✗ distance = SOUND_SPEED * 2 / echo_time

Objective 5 - A Little Resistance!

Time to create a warning system.

Now that you can detect the distance from the Rover to an object, you can create a distance detection warning system for the Rover.

Use the two LEDs you found from the old spacecraft for warning signals.

- You have two LEDs, but you don't know what color they are.
- **And** they look the same!

Let's get them connected and figure it out.

Concept: *LEDs and Resistors*

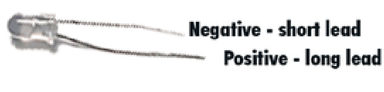
LEDs don't respond well when voltage fluctuates, and minor voltage increases can lead to significant increases in current. This can damage the LED.

A resistor is usually placed in series with the LED to limit the amount of current that passes through it.

- A resistor doesn't have a "front" or "back".
- It can be connected in any order.
- The resistor is connected to **power** and the LED.
- The LED is connected to the resistor and **ground**.

Each LED needs its own resistor.

- An LED has a **positive** lead and a **negative** lead.
- It must be connected in the correct order.
- The **positive** lead is longer and is connected to the resistor.
- The **negative** lead is shorter and is connected to **GND**.



Check the Hints to learn more about LEDs and resistors.

Run It!

CodeTrek:

```
1  from codex import *
2  import time
3  import pulseio
4
5  # Constants for peripherals
6  SOUND_SPEED = 34300     # Rate: centimeters / second
7  WAIT_TIME = 100         # Milliseconds
8  # TODO: Define LED_ON
9  # TODO: Define LED_OFF
```

Define constants for `LED_ON` and `LED_OFF`.

```
10
11 # Set up peripherals
12 echo = pulseio.PulseIn(exp.GPIO0)
13 trigger = exp.digital_out(exp.GPIO1)
14 red_led = exp.digital_out(exp.GPIO2)
```

Set up the LED as a digital output.

- This variable is for the **red** LED.
- In the next Objective you will set up another digital output for the **amber** LED.

```
15
16 # Convert echo time from microseconds to seconds
17 def convert_to_centimeters(echo_signal):
18     echo_time = echo_signal / 1000000
19     # Calculate using distance = rate * time
20     return (echo_time / 2) * SOUND_SPEED
21
22 # Use sonar to find the distance to an object
23 def object_distance():
24     # Turn on and off signal, clear the echo
25     trigger.value = True
26     trigger.value = False
27     echo.clear()
28     # Return time for echo or time out if no object detected
29     echo_start = time.ticks_ms()
30     while not echo:
31         if time.ticks_ms() > WAIT_TIME + echo_start:
32             return -1
33     distance_cm = convert_to_centimeters(echo[0])
34     return distance_cm
35
36 # Main program
37
38 # Test code for the red LED
39 red_led.value = LED_ON
40 time.sleep(2)
41 red_led.value = LED_OFF
```

Use this test code to turn on and off the LED.

- This will let you know everything is connected properly.
- You can also see the color of the LED.
- Try each LED, and leave in the one that is **red**.

```
42
43  while True:
44      distance_cm = object_distance()
45
46      if distance_cm < 0:
47          print("No object detected")
48      else:
49          print("Object detected:", distance_cm, "centimeters")
50
51      time.sleep_ms(WAIT_TIME)
```

Hints:

- ## Resistors and Ohms

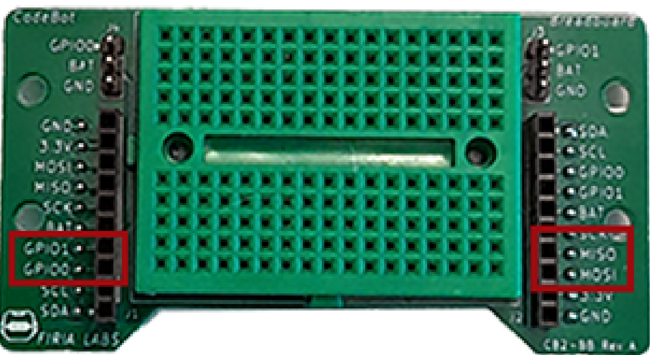
  Resistors are electronic components that limit the flow of electrons through a circuit. This is like a narrow pipe that restricts the flow of water.

  - A resistor is measured in **Ohms**.
  - The **Ohm** is a unit of resistance between two points in a conductor.
  - It is named after Georg Ohm, a Bavarian scientist who studied electricity.

  Resistors come in many different shapes, sizes and values. Most have colored bands printed on them that are used to indicate resistance value and tolerance.

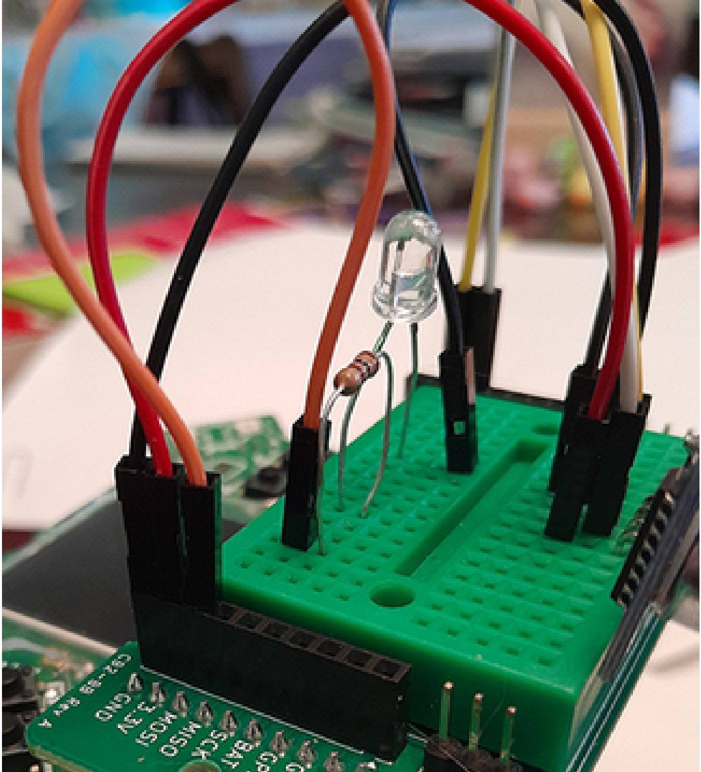
- ## Breadboard Input/Output

  It may look like the breadboard only has **two** input/output ports that are connected to the CodeX: **GPIO0** and **GPIO1**. But the breadboard actually has two more: MOSI and MISO. These are used for input/output as well, and are programmed as **GPIO2** and **GPIO3**.



- ## LED Test Connections

  After setting up the simple circuit with a resistor and LED, your breadboard may look like this:



Goals:

- Assign the 🔧constants `LED_ON = True` and `LED_OFF = False`.
- Set up the **red LED** by assigning the variable `red_led` as the output of `exp.digital_out(exp.GPIO2)`.
- Turn on the LED by assigning `red_led.value = LED_ON`.
- Turn off the LED by assigning `red_led.value = LED_OFF`.

Tools Found: Constants

Solution:

```
from codex import *
import time
import pulseio

# Constants for peripherals
SOUND_SPEED = 34300     # Rate: centimeters / second
WAIT_TIME = 100         # Milliseconds
LED_ON = True
LED_OFF = False

# Set up peripherals
echo = pulseio.PulseIn(exp.GPIO0)
trigger = exp.digital_out(exp.GPIO1)
red_led = exp.digital_out(exp.GPIO2)

# Convert echo time from microseconds to seconds
def convert_to_centimeters(echo_signal):
    echo_time = echo_signal / 1000000
    # Calculate using distance = rate * time
    return (echo_time / 2) * SOUND_SPEED

# Use sonar to find the distance to an object
def object_distance():
    # Turn on and off signal, clear the echo
```

```
25      trigger.value = True
26      trigger.value = False
27      echo.clear()
28      # Return time for echo or time out if no object detected
29      echo_start = time.ticks_ms()
30      while not echo:
31          if time.ticks_ms() > WAIT_TIME + echo_start:
32              return -1
33      distance_cm = convert_to_centimeters(echo[0])
34      return distance_cm
35
36  # Main program
37
38  # Test code for the red LED
39  red_led.value = LED_ON
40  time.sleep(2)
41  red_led.value = LED_OFF
42
43  while True:
44      distance_cm = object_distance()
45
46      if distance_cm < 0:
47          print("No object detected")
48      else:
49          print("Object detected:", distance_cm, "centimeters")
50
51      time.sleep_ms(WAIT_TIME)
52
```

Objective 6 **- Light It Up!**

Two LEDs are better than one!

You have the **red** LED connected, so follow the same steps to connect the **amber** LED.

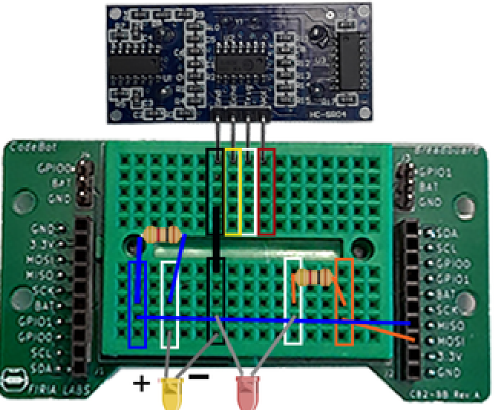
- It will need its own resistor and another jumper wire.
- It will connect to a different input/output connection point.

Physical Interaction: *LED and Resistor*

Keep all the connections for the ultrasonic sensor and red LED!

Create another simple circuit with one resistor, the **amber** LED and a new jumper wire.

- Connect a new jumper wire to **MISO** and a new terminal strip.
  - See the blue rectangle in the diagram.
- Connect one end of the resistor to the jumper wire (blue) terminal strip.
- Connect the other resistor end to a different terminal strip.
  - See the second white rectangle in the diagram.
- Connect the **amber** LED **positive** (long) lead to the same (white) terminal strip
- Connect the **amber** LED **negative** (short) lead to the **GND** (black) terminal strip



Make sure all wires are pushed into the breadbox for a solid connection.

Write some test code to make sure both LEDs are set up properly.

Check the 'Trek!

Follow the CodeTrek to test each LED.

- Set up the **amber** LED as a digital output.
- Turn on and off the **amber** LED.

Test your code to check the LEDs.

Run It!

Run the code. The **red** LED should light up for two seconds, followed by the **amber** LED for two seconds.

CodeTrek:

```
1  from codex import *
2  import time
3  import pulseio
4
5  # Constants for peripherals
6  SOUND_SPEED = 34300      # Rate: centimeters / second
7  WAIT_TIME = 100          # Milliseconds
8  LED_ON = True
9  LED_OFF = False
10
11 # Set up peripherals
12 echo = pulseio.PulseIn(exp.GPIO0)
13 trigger = exp.digital_out(exp.GPIO1)
14 red_led = exp.digital_out(exp.GPIO2)
15 # TODO: set up the amber_led as a digital output
```

Set up `amber_led` as a digital output.

- Use `GPIO3`

```
16
17 # Convert echo time from microseconds to seconds
18 def convert_to_centimeters(echo_signal):
19     echo_time = echo_signal / 1000000
20     # Calculate using distance = rate * time
21     return (echo_time / 2) * SOUND_SPEED
22
23 # Use sonar to find the distance to an object
24 def object_distance():
25     # Turn on and off signal, clear the echo
26     trigger.value = True
27     trigger.value = False
28     echo.clear()
29     # Return time for echo or time out if no object detected
30     echo_start = time.ticks_ms()
31     while not echo:
32         if time.ticks_ms() > WAIT_TIME + echo_start:
33             return -1
34     distance_cm = convert_to_centimeters(echo[0])
35     return distance_cm
36
37 # Main program
38
39 # Test code for the red LED
40 red_led.value = LED_ON
41 time.sleep(2)
42 red_led.value = LED_OFF
43 # Test code for the amber LED
44 amber_led.value = LED_ON
```

```
45  time.sleep(2)
46  amber_led.value = LED_OFF
```

Write test code to check that the **amber** LED is wired correctly.

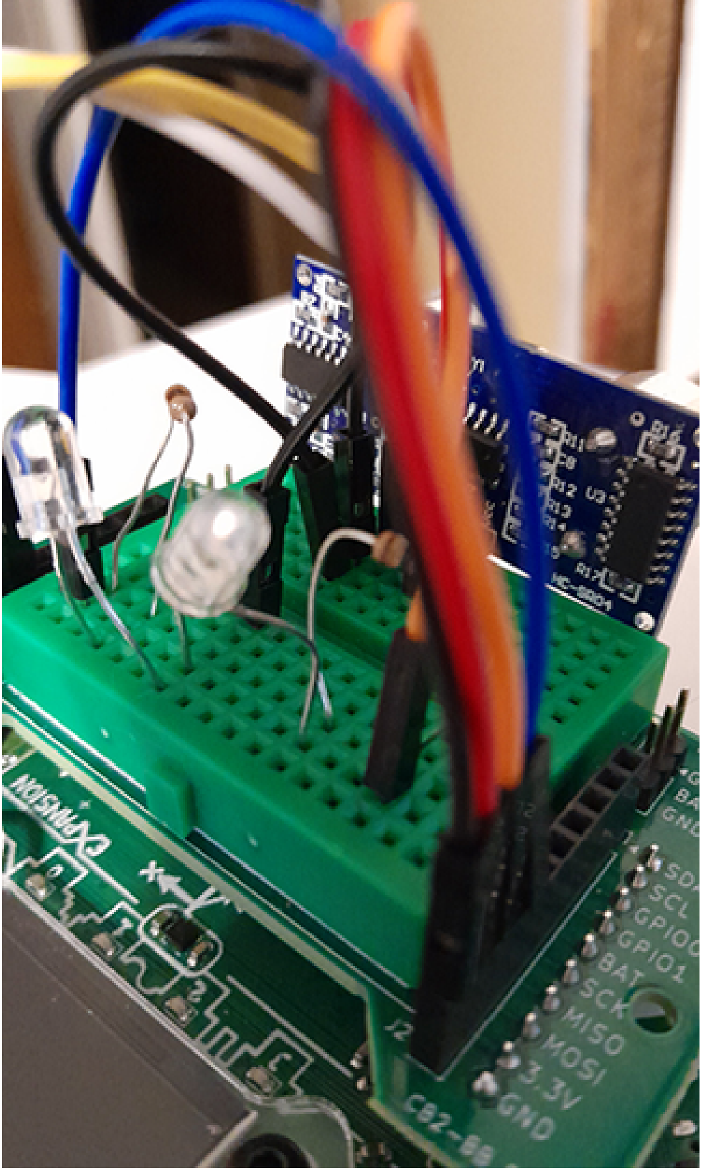
- The code is similar to the test code for the **red** LED.
- You will delete this *test code* in the next Objective.

```
47
48  while True:
49      distance_cm = object_distance()
50
51      if distance_cm < 0:
52          print("No object detected")
53      else:
54          print("Object detected:", distance_cm, "centimeters")
55
56      time.sleep_ms(WAIT_TIME)
```

Hint:

- ## LED Test Connections

  After setting up the second circuit with a resistor and **amber** LED, your breadboard may look like this:



Goals:

- Set up the **amber LED** by assigning the variable `amber_led` as the output of `exp.digital_out(exp.GPIO3)`.
- Turn on the amber LED by assigning `amber_led.value = LED_ON`.
- Turn off the amber LED by assigning `amber_led.value = LED_OFF`.

Solution:

```
1  from codex import *
2  import time
3  import pulseio
4  
5  # Constants for peripherals
6  SOUND_SPEED = 34300     # Rate: centimeters / second
7  WAIT_TIME = 100         # Milliseconds
8  LED_ON = True
9  LED_OFF = False
10 
11 # Set up peripherals
12 echo = pulseio.PulseIn(exp.GPIO0)
13 trigger = exp.digital_out(exp.GPIO1)
14 red_led = exp.digital_out(exp.GPIO2)
```

```
15 amber_led = exp.digital_out(exp.GPIO3)
16
17 # Convert echo time from microseconds to seconds
18 def convert_to_centimeters(echo_signal):
19     echo_time = echo_signal / 1000000
20     # Calculate using distance = rate * time
21     return (echo_time / 2) * SOUND_SPEED
22
23 # Use sonar to find the distance to an object
24 def object_distance():
25     # Turn on and off signal, clear the echo
26     trigger.value = True
27     trigger.value = False
28     echo.clear()
29     # Return time for echo or time out if no object detected
30     echo_start = time.ticks_ms()
31     while not echo:
32         if time.ticks_ms() > WAIT_TIME + echo_start:
33             return -1
34     distance_cm = convert_to_centimeters(echo[0])
35     return distance_cm
36
37 # Main program
38
39 # Test code for the red LED
40 red_led.value = LED_ON
41 time.sleep(2)
42 red_led.value = LED_OFF
43 # Test code for the amber LED
44 amber_led.value = LED_ON
45 time.sleep(2)
46 amber_led.value = LED_OFF
47
48 while True:
49     distance_cm = object_distance()
50
51     if distance_cm < 0:
52         print("No object detected")
53     else:
54         print("Object detected:", distance_cm, "centimeters")
55
56     time.sleep_ms(WAIT_TIME)
```

Objective 7 **- Danger Ahead!**

Use the two LEDs as warning signals for the Rover

The Rover's driver needs to be warned when a harmful object is near so that it can take the appropriate action, like turning or stopping.

- The **amber** LED can give a *warning* that the Rover is *approaching* an object.
- The **red** LED can give an *alert* that the Rover is *very close* to an object!

You have already designed alert systems for the spacecraft. This shouldn't be much different.

- Determine the distance to an object that triggers a *warning*.
- Determine the distance to an object that triggers an *alert*.
- Use an `if` and `elif` statement to turn on the **amber** or **red** LED.

Check the 'Trek!

Follow the CodeTrek to set up the distance detection warning system.

- Define constants for the **warning** and **alert** distances.
- Use an `if` and `elif` statement to turn *ON* the LEDs.

Test your code and check the warning system.

Run It!

Open the **console panel** and run the code.

- Start with an object far away from the sensor.
- Move the object close enough to the Rover to trigger the *warning*.
- Move the object closer to the Rover to trigger the *alert*.
- Now move the object farther away.

What did you notice about the alarm system? Does it work the way you expect it to?

CodeTrek:

```
1  from codex import *
2  import time
3  import pulseio
4
5  # Constants for peripherals
6  SOUND_SPEED = 34300      # Rate: centimeters / second
7  WAIT_TIME = 100          # Milliseconds
8  LED_ON = True
9  LED_OFF = False
10 WARNING_DISTANCE = 20.0 # Centimeters
11 ALERT_DISTANCE = 10.0   # Centimeters
```

Define constants for `WARNING_DISTANCE` and `ALERT_DISTANCE`.

- For testing purposes, the distances will be fairly small.
- The suggested distances are `20.0` and `10.0` but you can use different values if you want.
- Using constants for the distances makes it easy to change the values when deploying the system for the actual Rover.

```
12
13 # Set up peripherals
14 echo = pulseio.PulseIn(exp.GPIO0)
15 trigger = exp.digital_out(exp.GPIO1)
16 red_led = exp.digital_out(exp.GPIO2)
17 amber_led = exp.digital_out(exp.GPIO3)
18
19 # Convert echo time from microseconds to seconds
20 def convert_to_centimeters(echo_signal):
21     echo_time = echo_signal / 1000000
22     # Calculate using distance = rate * time
23     return (echo_time / 2) * SOUND_SPEED
24
25 # Use sonar to find the distance to an object
26 def object_distance():
27     # Turn on and off signal, clear the echo
28     trigger.value = True
29     trigger.value = False
30     echo.clear()
31     # Return time for echo or time out if no object detected
32     echo_start = time.ticks_ms()
33     while not echo:
34         if time.ticks_ms() > WAIT_TIME + echo_start:
35             return -1
36     distance_cm = convert_to_centimeters(echo[0])
37     return distance_cm
38
39 # Main program
40
```

Delete all the test code used to check the **red** and **amber** LEDs.

```
41 while True:
42     distance_cm = object_distance()
43
44     if distance_cm < 0:
45         print("No object detected")
```

```
46      else:
47          print("Object detected:", distance_cm, "centimeters")
48
49          if distance_cm < ALERT_DISTANCE:
```

Add the new `if` statement inside the `else` section of the first `if` statement.

- You only need to check the proximity of an object if an object is actually detected.
- The order of the conditions matters!
  - Always start with the most restrictive condition.
  - In this case, it is the smallest distance, or the **alert**.

```
50              red_led.value = LED_ON
```

If the Rover is *very close* to an object, turn *ON* the **red** LED.

```
51          elif distance_cm < WARNING_DISTANCE:
52              amber_led.value = LED_ON
```

Add an `elif` to the statement that checks for the *warning* distance.

- If the Rover is *approaching* an object, turn *ON* the **amber** LED.

```
53
54      time.sleep_ms(WAIT_TIME)
```

Hint:

- ## Danger Distances

  How can you determine the distances to use for the warning system?

  The distances used for testing are somewhat arbitrary.

  - You are using the CodeX as the Rover, and the CodeX is small.
  - Distance to an object is measured in centimeters.
  - Select a *warning* distance that seems far enough to keep going, but close enough to be cautious.
  - Select an *alert* distance that seems almost too close for the Rover to stop safely.

Goals:

- Define the constants `WARNING_DISTANCE` and `ALERT_DISTANCE`.
- Add an `if` statement with the condition `distance_cm < ALERT_DISTANCE`.
- Turn *ON* the **red** LED as an *alert*.
- Add in `elif` statement that checks if `distance_cm < WARNING_DISTANCE`.
- Turn *ON* the **amber** LED as a *warning*.

Tools Found: Constants, bool

Solution:

```
1  from codex import *
2  import time
3  import pulseio
4
5  # Constants for peripherals
6  SOUND_SPEED = 34300      # Rate: centimeters / second
7  WAIT_TIME = 100          # Milliseconds
8  LED_ON = True
```

```
9   LED_OFF = False
10  WARNING_DISTANCE = 20.0 # Centimeters
11  ALERT_DISTANCE = 10.0   # Centimeters
12
13  # Set up peripherals
14  echo = pulseio.PulseIn(exp.GPIO0)
15  trigger = exp.digital_out(exp.GPIO1)
16  red_led = exp.digital_out(exp.GPIO2)
17  amber_led = exp.digital_out(exp.GPIO3)
18
19  # Convert echo time from microseconds to seconds
20  def convert_to_centimeters(echo_signal):
21      echo_time = echo_signal / 1000000
22      # Calculate using distance = rate * time
23      return (echo_time / 2) * SOUND_SPEED
24
25  # Use sonar to find the distance to an object
26  def object_distance():
27      # Turn on and off signal, clear the echo
28      trigger.value = True
29      trigger.value = False
30      echo.clear()
31      # Return time for echo or time out if no object detected
32      echo_start = time.ticks_ms()
33      while not echo:
34          if time.ticks_ms() > WAIT_TIME + echo_start:
35              return -1
36      distance_cm = convert_to_centimeters(echo[0])
37      return distance_cm
38
39  # Main program
40
41  while True:
42      distance_cm = object_distance()
43
44      if distance_cm < 0:
45          print("No object detected")
46      else:
47          print("Object detected:", distance_cm, "centimeters")
48
49          if distance_cm < ALERT_DISTANCE:
50              red_led.value = LED_ON
51          elif distance_cm < WARNING_DISTANCE:
52              amber_led.value = LED_ON
53
54      time.sleep_ms(WAIT_TIME)
```

Quiz 2 - Check Your Understanding: LEDs and Resistors

Question 1: Why does an LED need a resistor?

✓ To limit the voltage.

✗ To increase the voltage.

✗ To connect it to the breadboard.

✗ To receive data from the sensor.

Question 2: Which LED **lead** is **positive**?

✓ The longer side

✗ The shorter side

✗ Either one; it doesn't matter

✗ The **red** one

Question 3: What does the LED **negative lead** connect to?

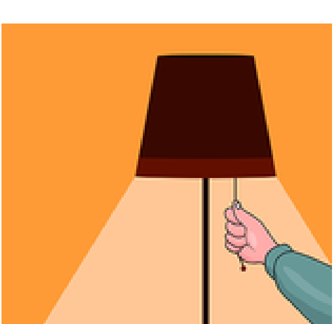
- ✓ GND
- ✗ 3.3V
- ✗ The resistor
- ✗ Input/Output

Question 4: When setting up the warning system, which condition comes first?

- ✓ Check for the *alert* distance.
- ✗ Check for the *warning* distance.
- ✗ Check if the Rover isn't close to the *warning* or *alert* distances.
- ✗ Any of the conditions can be first.

Objective 8 - Turn the Lights Off!

The warning system works ... the first time



The **warning** and **alert** LEDs light up, but when the Rover is no longer near an object, the LEDs stay on. And, you don't want the **warning** LED and the **alert** LED to show at the same time.

Let's fix this problem.

- When the **amber** LED is *ON*, the **red** LED should be *OFF*.
- When the **red** LED is *ON*, the **amber** LED should be *OFF*.
- Both LEDs should be *OFF* if the Rover is not close to any harmful object.

Check the 'Trek!

Follow the CodeTrek to add code to the `if` and `elif` statement.

- Turn *OFF* the **amber** LED when the **red** LED is *ON*.
- Turn *OFF* the **red** LED when the **amber** LED is *ON*.
- Turn *OFF* both LEDs when neither condition is `True`.

Test your code and re-check the warning system.

Run It!

Open the **console panel** and run the code.

- Start with an object far away from the sensor.
- Move the object close enough to the Rover to trigger the *warning*.
- Move the object closer to the Rover to trigger the *alert*.
- Now move the object farther away.

Now does the warning system work the way you expect it to?

CodeTrek:

```
1  from codex import *
2  import time
3  import pulseio
4
5  # Constants for peripherals
```

```
6  SOUND_SPEED = 34300     # Rate: centimeters / second
7  WAIT_TIME = 100         # Milliseconds
8  LED_ON = True
9  LED_OFF = False
10 WARNING_DISTANCE = 20.0 # Centimeters
11 ALERT_DISTANCE = 10.0   # Centimeters
12 
13 # Set up peripherals
14 echo = pulseio.PulseIn(exp.GPIO0)
15 trigger = exp.digital_out(exp.GPIO1)
16 red_led = exp.digital_out(exp.GPIO2)
17 amber_led = exp.digital_out(exp.GPIO3)
18 
19 # Convert echo time from microseconds to seconds
20 def convert_to_centimeters(echo_signal):
21     echo_time = echo_signal / 1000000
22     # Calculate using distance = rate * time
23     return (echo_time / 2) * SOUND_SPEED
24 
25 # Use sonar to find the distance to an object
26 def object_distance():
27     # Turn on and off signal, clear the echo
28     trigger.value = True
29     trigger.value = False
30     echo.clear()
31     # Return time for echo or time out if no object detected
32     echo_start = time.ticks_ms()
33     while not echo:
34         if time.ticks_ms() > WAIT_TIME + echo_start:
35             return -1
36     distance_cm = convert_to_centimeters(echo[0])
37     return distance_cm
38 
39 # Main program
40 while True:
41     distance_cm = object_distance()
42 
43     if distance_cm < 0:
44         print("No object detected")
45     else:
46         print("Object detected:", distance_cm, "centimeters")
47 
48         if distance_cm < ALERT_DISTANCE:
49             red_led.value = LED_ON
50             amber_led.value = LED_OFF
```

If `distance_cm < ALERT_DISTANCE` turn the **amber** LED *OFF*.

```
51         elif distance_cm < WARNING_DISTANCE:
52             amber_led.value = LED_ON
53             red_led.value = LED_OFF
```

`elif distance_cm < WARNING_DISTANCE` turn the **red** LED *OFF*.

```
54         else:
55             red_led.value = LED_OFF
56             amber_led.value = LED_OFF
```

`else` if neither condition is `True`, turn *OFF* both LEDs.

```
57 
58     time.sleep_ms(WAIT_TIME)
```

Goals:

- Use an `if` statement with the condition `distance_cm < ALERT_DISTANCE`.
- When triggered, turn *OFF* the **amber** LED.

- Use an `elif` statement with the condition `distance_cm < WARNING_DISTANCE`.
- When triggered, turn *OFF* the **red** LED.
- Use an `else` statement.
- When triggered, turn *OFF* both LEDs.

Tools Found: bool

Solution:

```
from codex import *
import time
import pulseio

# Constants for peripherals
SOUND_SPEED = 34300     # Rate: centimeters / second
WAIT_TIME = 100         # Milliseconds
LED_ON = True
LED_OFF = False
WARNING_DISTANCE = 20.0 # Centimeters
ALERT_DISTANCE = 10.0   # Centimeters

# Set up peripherals
echo = pulseio.PulseIn(exp.GPIO0)
trigger = exp.digital_out(exp.GPIO1)
red_led = exp.digital_out(exp.GPIO2)
amber_led = exp.digital_out(exp.GPIO3)

# Convert echo time from microseconds to seconds
def convert_to_centimeters(echo_signal):
    echo_time = echo_signal / 1000000
    # Calculate using distance = rate * time
    return (echo_time / 2) * SOUND_SPEED

# Use sonar to find the distance to an object
def object_distance():
    # Turn on and off signal, clear the echo
    trigger.value = True
    trigger.value = False
    echo.clear()
    # Return time for echo or time out if no object detected
    echo_start = time.ticks_ms()
    while not echo:
        if time.ticks_ms() > WAIT_TIME + echo_start:
            return -1
    distance_cm = convert_to_centimeters(echo[0])
    return distance_cm

# Main program
while True:
    distance_cm = object_distance()

    if distance_cm < 0:
        print("No object detected")
    else:
        print("Object detected:", distance_cm, "centimeters")

        if distance_cm < ALERT_DISTANCE:
            red_led.value = LED_ON
            amber_led.value = LED_OFF
        elif distance_cm < WARNING_DISTANCE:
            amber_led.value = LED_ON
            red_led.value = LED_OFF
        else:
            red_led.value = LED_OFF
            amber_led.value = LED_OFF

    time.sleep_ms(WAIT_TIME)
```

<u>*Objective 9* - Add Color and Sound</u>

Use the CodeX for extra warning system features.

So far you have used the salvaged parts from the long-lost spacecraft for the distance detection warning system. But some parts of the CodeX are still functional. You can add some of the features of the CodeX into the warning system.

First Feature: Display Color

You discover the CodeX display screen can still display color. This is great news! Change the display screen:

- **RED** when the *alert* is triggered,
- **YELLOW** when the *warning* is triggered,
- **GREEN** when nothing is triggered,
- and show an image from the pics library when no obstacle is detected.

Type in the Code

Add code to each of the condition blocks to display something on the screen. *Be careful with indenting!*

- if no obstacle is detected, select an image to display, like the happy face.

```
display.show(pics.HAPPY)
```

- If the *alert* is triggered, change the display to **RED**.

```
display.fill(RED)
```

- If the *warning* is triggered, change the display to **YELLOW**.

```
display.fill(YELLOW)
```

- `else` change the display to **GREEN**.

```
display.fill(GREEN)
```

Test the warning system's added feature.

Run It!

Run the code. Check all the possibilities and make sure the correct image or color displays on the screen.

Second Feature: Add a warning sound

You also find out the speaker still works on the CodeX. Add to the visual warning system by adding an audio component as well.

- Make a high-pitched siren beep when the *alert* is triggered.
- Make a lower-pitched siren beep when the *warning* is triggered.
- Don't make any sound when the Rover is not in danger.

Check the Hints to review the soundlib module.

There are several steps for creating a siren. So ... write a function!

Check the 'Trek!

Follow the CodeTrek to define a function for the siren and call the function when a *warning* or *alert* is triggered.

Test your code and check the new features to the warning system.

Run It!

Run the code. Check all the possibilities and make sure:

- The correct image or color displays on the screen.
- The siren plays when a *warning* or *alert* is triggered.

CodeTrek:

```
1  from codex import *
2  import time
3  import pulseio
4  from soundlib import *
5
6  siren = soundmaker.get_tone("violin")
```

Define a variable for siren.

- Remember to import the `soundlib` module.

```
7
8  # Constants for peripherals
9  SOUND_SPEED = 34300     # Rate: centimeters / second
10 WAIT_TIME = 100         # Milliseconds
11 LED_ON = True
12 LED_OFF = False
13 WARNING_DISTANCE = 20.0 # Centimeters
14 ALERT_DISTANCE = 10.0   # Centimeters
15
16 # Set up peripherals
17 echo = pulseio.PulseIn(exp.GPIO0)
18 trigger = exp.digital_out(exp.GPIO1)
19 red_led = exp.digital_out(exp.GPIO2)
20 amber_led = exp.digital_out(exp.GPIO3)
21
22 # Convert echo time from microseconds to seconds
23 def convert_to_centimeters(echo_signal):
24     echo_time = echo_signal / 1000000
25     # Calculate using distance = rate * time
26     return (echo_time / 2) * SOUND_SPEED
27
28 # Use sonar to find the distance to an object
29 def object_distance():
30     # Turn on and off signal, clear the echo
31     trigger.value = True
32     trigger.value = False
33     echo.clear()
34     # Return time for echo or time out if no object detected
35     echo_start = time.ticks_ms()
36     while not echo:
37         if time.ticks_ms() > WAIT_TIME + echo_start:
38             return -1
39     distance_cm = convert_to_centimeters(echo[0])
40     return distance_cm
41
42 # Sound a siren when the Rover is in danger
43 def sound_siren(tone):
```

Define a function for sounding the siren.

- Use **one parameter** for the tone.
- The parameter allows you to use a single function for **two** different sirens.

```
44     siren.set_pitch(tone)
45     siren.play()
46     siren.glide(tone+440, 2.0)
47     time.sleep_ms(WAIT_TIME)
48     siren.stop()
```

Add code to sound the siren.

- The starting tone is the value of the parameter.
- The `glide` gives a nice siren sound.

- As long as the condition in the **main program** is `True`, the function will be called continously, which will make the siren "beep".

```
49
50  # Main program
51  while True:
52      distance_cm = object_distance()
53
54      if distance_cm < 0:
55          print("No object detected")
56          display.show(pics.HAPPY)
```

Add code to display the happy face when no obstacle is detected.

- You can use a different image, or even a fill color.
- The code is from the first feature.

```
57      else:
58          print("Object detected:", distance_cm, "centimeters")
59
60          if distance_cm < ALERT_DISTANCE:
61              red_led.value = LED_ON
62              amber_led.value = LED_OFF
63              display.fill(RED)
64              sound_siren(770)
65          elif distance_cm < WARNING_DISTANCE:
66              amber_led.value = LED_ON
67              red_led.value = LED_OFF
68              display.fill(YELLOW)
69              sound_siren(440)
```

Call the function if the alert or warning is triggered.

- You will have **two** function calls.
- Use a different tone for each function call.

```
70          else:
71              red_led.value = LED_OFF
72              amber_led.value = LED_OFF
73              display.fill(GREEN)
```

Add code to each of the `if` statement branches to display
a fill color on the screen.

- You will have three of these statements, one for each condition block.
- The code is from the first feature.

```
74
75      time.sleep_ms(WAIT_TIME)
```

Hints:

- ## The Soundlib Module

  The soundlib module (or library) provides a high-level interface for playing and mixing sounds of different types on the CodeX.

  A first step for using methods in the soundlib module is to define a variable for an instrument. You can choose from:

  - flute
  - recorder
  - trumpet
  - violin
  - noise
  - organ
  - synth

Experiment with the different instruments and pick one you like the best.

- ## Gliding

  A feature of the 🔧soundlib module is **glide**.

  - This method takes **two parameters**: the ending pitch and the duration.
  - The sound will glide from the current frequency to the new pitch.
  - The timing of the glide is the duration in seconds.

- ## Make It Your Own

  The added features offer a lot of flexibility. You can make some decisions so that the warning system is uniquely yours.

  You can:

  - select the colors and/or images to display on the screen.
  - pick an instrument for the siren.
  - choose the starting pitch (tone) for each warning beep.
  - choose the ending pitch for the glide.
  - determine the duration of the glide. This can even be a parameter and can be different for each function call.
  - choose to call the siren even if no obstacle is detected.
  - select the warning and alert distance, and also the wait time.

Goals:

- Use an `if` statement with the 🔧condition `distance_cm < ALERT_DISTANCE`.

  When triggered:

- Call `display.fill(RED)`.
- Call `sound_siren(770)`.
- Use an `elif` statement with the 🔧condition `distance_cm < WARNING_DISTANCE`.

  When triggered:

- Call `display.fill(YELLOW)`.
- Call `sound_siren(440)`.
- Define a function `sound_siren()` that has one parameter `tone`.

Tools Found: Display, CodeX Image Pics, Audio, soundlib, Functions, bool

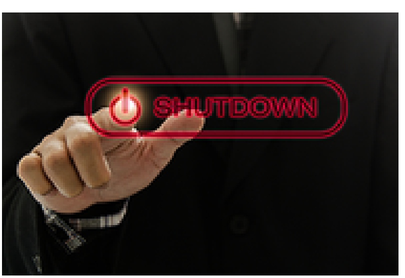
Solution:

```
1  from codex import *
2  import time
3  import pulseio
4  from soundlib import *
5
6  siren = soundmaker.get_tone("violin")
7
8  # Constants for peripherals
9  SOUND_SPEED = 34300      # Rate: centimeters / second
10 WAIT_TIME = 100          # Milliseconds
11 LED_ON = True
12 LED_OFF = False
13 WARNING_DISTANCE = 20.0 # Centimeters
14 ALERT_DISTANCE = 10.0   # Centimeters
15
```

```
16 # Set up peripherals
17 echo = pulseio.PulseIn(exp.GPIO0)
18 trigger = exp.digital_out(exp.GPIO1)
19 red_led = exp.digital_out(exp.GPIO2)
20 amber_led = exp.digital_out(exp.GPIO3)
21
22 # Convert echo time from microseconds to seconds
23 def convert_to_centimeters(echo_signal):
24     echo_time = echo_signal / 1000000
25     # Calculate using distance = rate * time
26     return (echo_time / 2) * SOUND_SPEED
27
28 # Use sonar to find the distance to an object
29 def object_distance():
30     # Turn on and off signal, clear the echo
31     trigger.value = True
32     trigger.value = False
33     echo.clear()
34     # Return time for echo or time out if no object detected
35     echo_start = time.ticks_ms()
36     while not echo:
37         if time.ticks_ms() > WAIT_TIME + echo_start:
38             return -1
39     distance_cm = convert_to_centimeters(echo[0])
40     return distance_cm
41
42 # Sound a siren when the Rover is in danger
43 def sound_siren(tone):
44     siren.set_pitch(tone)
45     siren.play()
46     siren.glide(tone+440, 2.0)
47     time.sleep_ms(WAIT_TIME)
48     siren.stop()
49
50 # Main program
51 while True:
52     distance_cm = object_distance()
53
54     if distance_cm < 0:
55         print("No object detected")
56         display.show(pics.HAPPY)
57     else:
58         print("Object detected:", distance_cm, "centimeters")
59
60         if distance_cm < ALERT_DISTANCE:
61             red_led.value = LED_ON
62             amber_led.value = LED_OFF
63             display.fill(RED)
64             sound_siren(770)
65         elif distance_cm < WARNING_DISTANCE:
66             amber_led.value = LED_ON
67             red_led.value = LED_OFF
68             display.fill(YELLOW)
69             sound_siren(440)
70         else:
71             red_led.value = LED_OFF
72             amber_led.value = LED_OFF
73             display.fill(GREEN)
74
75     time.sleep_ms(WAIT_TIME)
```

Objective 10 **- Power Down!**

And one more thing ...



Did you notice that if an LED is *ON* when the program ends, it *STAYS* on? This can be a little annoying. Plus, you just might want the ability to power down the Rover at any given time.

The CodeX can really help out with this feature. Use a CodeX Button to "power down" the Rover.

Check the 'Trek!

Follow the CodeTrek to define a `power_down` function and call it when a button is pressed.

Test your code and power down the Rover on demand.

Run It!

Run the code.

- Check all the features of the warning system.
- Power down the Rover by pressing the button.

CodeTrek:

```
1  from codex import *
2  import time
3  import pulseio
4  from soundlib import *
5
6  siren = soundmaker.get_tone("violin")
7
8  # Constants for peripherals
9  SOUND_SPEED = 34300     # Rate: centimeters / second
10 WAIT_TIME = 100         # Milliseconds
11 LED_ON = True
12 LED_OFF = False
13 WARNING_DISTANCE = 20.0 # Centimeters
14 ALERT_DISTANCE = 10.0   # Centimeters
15
16 # Set up peripherals
17 echo = pulseio.PulseIn(exp.GPIO0)
18 trigger = exp.digital_out(exp.GPIO1)
19 red_led = exp.digital_out(exp.GPIO2)
20 amber_led = exp.digital_out(exp.GPIO3)
21
22 # Convert echo time from microseconds to seconds
23 def convert_to_centimeters(echo_signal):
24     echo_time = echo_signal / 1000000
25     # Calculate using distance = rate * time
26     return (echo_time / 2) * SOUND_SPEED
27
28 # Use sonar to find the distance to an object
29 def object_distance():
30     # Turn on and off signal, clear the echo
31     trigger.value = True
32     trigger.value = False
33     echo.clear()
34     # Return time for echo or time out if no object detected
35     echo_start = time.ticks_ms()
36     while not echo:
37         if time.ticks_ms() > WAIT_TIME + echo_start:
38             return -1
39     distance_cm = convert_to_centimeters(echo[0])
40     return distance_cm
41
42 # Sound a siren when the Rover is in danger
43 def sound_siren(tone):
44     siren.set_pitch(tone)
45     siren.play()
46     siren.glide(tone+440, 2.0)
47     time.sleep_ms(WAIT_TIME)
48     siren.stop()
49
50 # Turn off the warning system and stop the Rover
51 def power_down():
```

Define the function `power_down()` to shut off the warning system.

```
52    red_led.value = LED_OFF
53    amber_led.value = LED_OFF
```

Turn *OFF* both LEDs, in case they are *ON* when you want to power down the system.

```
54    print("Rover stopped")
55    display.show(pics.HOUSE)
```

Indicate on the **console panel** and the display screen that the Rover has stopped.

- You can select any image from the pics library to display.

```
56    time.sleep(5)
```

Set a delay for the `print()` and `display.show()` statements.

- After the delay they will automatically turn off.

```
57
58 # Main program
59 while True:
60     distance_cm = object_distance()
61
62     if distance_cm < 0:
63         print("No object detected")
64         display.show(pics.HAPPY)
65     else:
66         print("Object detected:", distance_cm, "centimeters")
67
68         if distance_cm < ALERT_DISTANCE:
69             red_led.value = LED_ON
70             amber_led.value = LED_OFF
71             display.fill(RED)
72             sound_siren(770)
73         elif distance_cm < WARNING_DISTANCE:
74             amber_led.value = LED_ON
75             red_led.value = LED_OFF
76             display.fill(YELLOW)
77             sound_siren(440)
78         else:
79             red_led.value = LED_OFF
80             amber_led.value = LED_OFF
81             display.fill(GREEN)
82
83     time.sleep_ms(WAIT_TIME)
84
85     if buttons.was_pressed(BTN_A):
86         break
```

In the `while True:` loop, use a button press to break from the loop.

- Be careful with indenting!

```
87
88 # End program
89 power_down()
```

After the `while True:` loop, end the program by calling `power_down()`.

- The function call is **after** the loop.
- Be **very careful** with indenting!

Hints:

- ## The Power of a Function

  Since you are only calling the `power_down()` function **once**, you may wonder why you need to define a function at all. You could easily just put all the code at the end.

  This is true, but defining a function gives you a lot of flexibility. Let's say you add a servo for wheels on the Rover. When you **power down** you also want to **turn off** the wheels. A function keeps code all in one place and makes it easy to modify.

- ## Inputs

  You are using a CodeX button to break the loop and subsequently power down the Rover. What if the CodeX buttons are non-functioning? You could also use a peripheral from the Kit, such as:

  - the button
  - the switch
  - the microswitch

  You could also use a sensor. If a particular reading is given, you may want to call the `power_down()` function.

  - temperature sensor, if the Rover gets too hot or cold
  - light sensor, if it is too dark for the Rover to see
  - sound sensor, if there is an explosion on the Rover
  - and many more possibilities!

Goals:

- Define the function `power_down()`.
- Use an `if` statement with the condition `buttons.was_pressed(BTN_A)`.
- When triggered, call `break`.
- Call `power_down()`.

Tools Found: CodeX Buttons, Functions, bool, CodeX Image Pics, Timing

Solution:

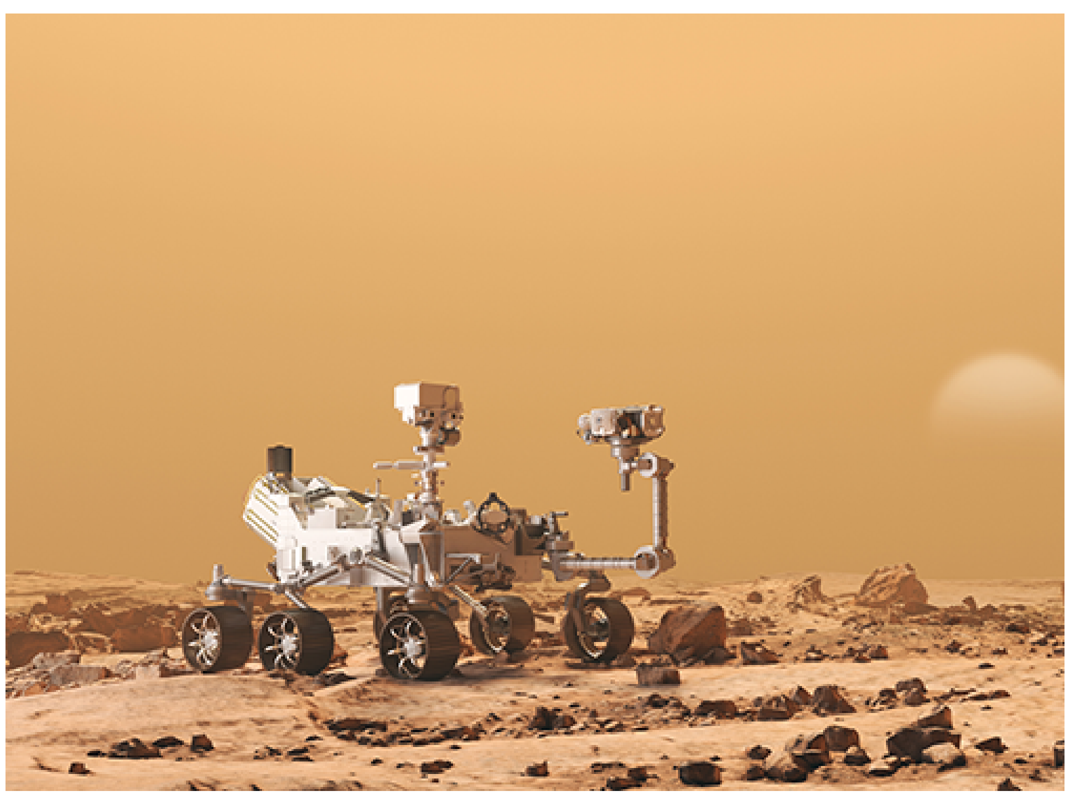
```
1  from codex import *
2  import time
3  import pulseio
4  from soundlib import *
5
6  siren = soundmaker.get_tone("violin")
7
8  # Constants for peripherals
9  SOUND_SPEED = 34300     # Rate: centimeters / second
10 WAIT_TIME = 100         # Milliseconds
11 LED_ON = True
12 LED_OFF = False
13 WARNING_DISTANCE = 20.0 # Centimeters
14 ALERT_DISTANCE = 10.0   # Centimeters
15
16 # Set up peripherals
17 echo = pulseio.PulseIn(exp.GPIO0)
18 trigger = exp.digital_out(exp.GPIO1)
```

```
19 red_led = exp.digital_out(exp.GPIO2)
20 amber_led = exp.digital_out(exp.GPIO3)
21
22 # Convert echo time from microseconds to seconds
23 def convert_to_centimeters(echo_signal):
24     echo_time = echo_signal / 1000000
25     # Calculate using distance = rate * time
26     return (echo_time / 2) * SOUND_SPEED
27
28 # Use sonar to find the distance to an object
29 def object_distance():
30     # Turn on and off signal, clear the echo
31     trigger.value = True
32     trigger.value = False
33     echo.clear()
34     # Return time for echo or time out if no object detected
35     echo_start = time.ticks_ms()
36     while not echo:
37         if time.ticks_ms() > WAIT_TIME + echo_start:
38             return -1
39     distance_cm = convert_to_centimeters(echo[0])
40     return distance_cm
41
42 # Sound a siren when the Rover is in danger
43 def sound_siren(tone):
44     siren.set_pitch(tone)
45     siren.play()
46     siren.glide(tone+440, 2.0)
47     time.sleep_ms(WAIT_TIME)
48     siren.stop()
49
50 # Turn off the warning system and stop the Rover
51 def power_down():
52     red_led.value = LED_OFF
53     amber_led.value = LED_OFF
54     print("Rover stopped")
55     display.show(pics.HOUSE)
56     time.sleep(5)
57
58 # Main program
59 while True:
60     distance_cm = object_distance()
61
62     if distance_cm < 0:
63         print("No object detected")
64         display.show(pics.HAPPY)
65     else:
66         print("Object detected:", distance_cm, "centimeters")
67
68         if distance_cm < ALERT_DISTANCE:
69             red_led.value = LED_ON
70             amber_led.value = LED_OFF
71             display.fill(RED)
72             sound_siren(770)
73         elif distance_cm < WARNING_DISTANCE:
74             amber_led.value = LED_ON
75             red_led.value = LED_OFF
76             display.fill(YELLOW)
77             sound_siren(440)
78         else:
79             red_led.value = LED_OFF
80             amber_led.value = LED_OFF
81             display.fill(GREEN)
82
83     time.sleep_ms(WAIT_TIME)
84
85     if buttons.was_pressed(BTN_A):
86         break
87
88 # End program
89 power_down()
```

Mission 10 Complete

You've completed Project Exploring the Surface!

The Rover is now less likely to crash. And the crew can focus on their mission instead of fixing the equipment.



And your mission here is now complete!

Your next mission, should you choose to accept it, is to use your programming skills and the peripherals kit to complete your own projects.

Have fun, and happy coding!