Lab 3: **Embedded Microcontroller Signals and Github**

SED 1115 – Embedded Programming and Scripting

Faculty of Engineering – University of Ottawa

Lab Objectives:

The basic purpose of this lab is to learn how to use the GPIO pins of a Raspberry Pi Pico to perform signal input and output. Specifically, this will be: digital input and digital output, analog input, and digital PWM output. The lab will also cover how to use git and GitHub to manage your code. Using git or github will allow you to work better on coding projects with other people. You can use the tool to manage and share your code with others, as you continue to update it over time.

Lab Instructions:

Parts and Equipment

You will need: a Raspberry Pi Pico (or Pico, for short!), a laptop or other computer with USB interfaces and internet access, a USB micro-B cable that works with that computer and also an SED1115-custom-designed "expansion" board.

Warnings

Note that a Pico is a 3.3V device that should not be connected to 5V devices or to other potential 5V power sources, other than via the onboard USB port (which is a supported 5V standard by the Pico). Connecting the Pico directly to other 5V devices can damage it. Please do not use the Pico on a conductive surface (e.g. metal table) to avoid shorting the exposed pins on the bottom of the device. Even when mounted on the "accessory" board, which has included mechanical standoff posts, shorting pins might still be possible, if the surface is very uneven.

It is recommended that you place the Pico (and/or the expansion board) on a flat, non-conductive, anti-static surface, such as a wooden table or a notebook. Electrostatic discharge (ESD) can damage the pico (or most other electronic devices). To avoid such damage, when working on the Pico, ground yourself using an approved grounded ESD wrist strap (properly connected to ground) or touch a grounded metal part of a grounded object (e.g. your computer case), constantly, while working with ***any*** ESD-sensitive semiconductor devices. Try to touch only the metal standoffs of the accessory board.

GPIO Pins

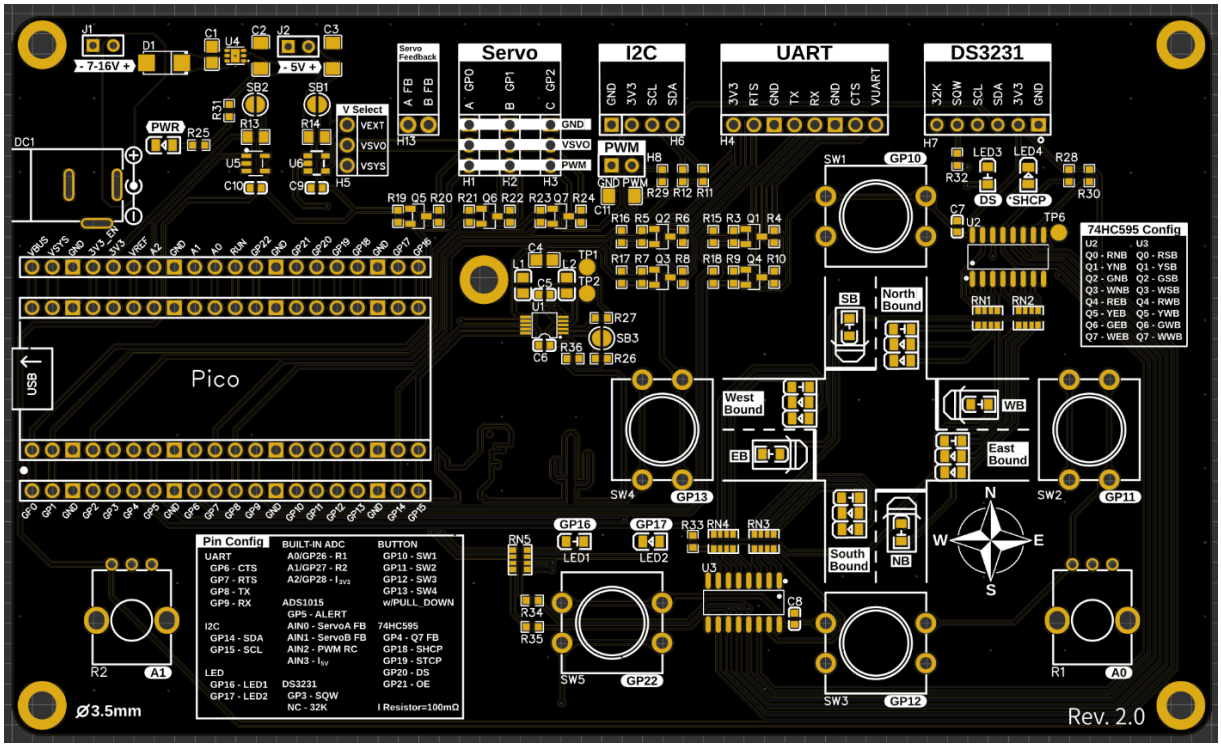
The Pico has 26 multifunctional General Purpose Input / Output (GPIO) pins. These pins can be configured via software to be used for digital input or digital output, analog input, PWM output or I2C/ SPI/UART serial communications links and more.

The number of pins that can be used for each function is limited by the number of hardware blocks that are available for each function. For example, there are only 2 hardware I2C channels on the Pico, so you can only use 2 sets of I2C pins at a time. Use the interactive pinout diagram for Pico or Pico W to see which pins can be used for each function. The Pico W is the recommended device for this course, with a "W" suffix used to indicate the presence of a **W**ireless communications interface.

Expansion Board

For the course, there is a custom-designed expansion board for the Raspberry Pi Pico, featuring 5 buttons, 2 potentiometers, several LEDs, and bi-directional level shifters. It has breakout pins for three "Servo" PWM channels, an I2C channel, a UART channel (with level shifters on RX, TX, RTS, and CTS to allow interfacing to a 5V device like an Arduino), and a socket for a DS3231 RTC (Real-Time Controller) module. It also supports external power input for the PWM channels and the level shifters, although there should not be a need to use these features in this lab. There is also a representation of a 4-way traffic intersection (with traffic light LEDs and an indicator for car presence and a button nearby to simulate a car arrival event). Many of these will be used in future labs and project work.

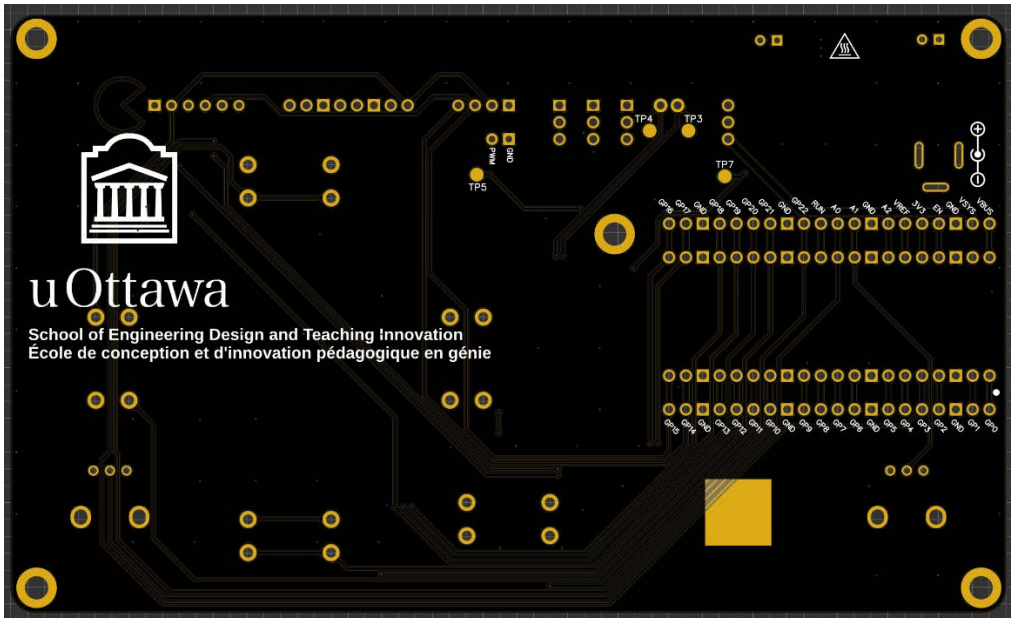
When inserting the Pico into the expansion board, make sure the Pico is facing in the proper direction. The USB port should be facing away from the LEDs (and outwards from the centre of the board) as the white silkscreen on the accessory board indicates. Insert the Pico carefully, so as not to bend any pins under the device or in a way that makes them splay outwards either. Each pin must go into its appropriate socket, so make sure that the device is aligned properly and not offset by a pin position, etc.



Primary side of the Accessory Board

Connections to the Pico are printed on the silkscreen in front of the board and included here, for reference purposes.

Designation	GPIO	Functional Description
SHCP	GP18	Shift Register Input Clock
STCP	GP19	Shift Register Output Storage Clock
DS	GP20	Shift Register Serial Data Input
OE	GP21	Shift Register Output Enable
Q7FB	GP4	Shift Register Final Stage Output
SW1	GP10	SW 1 Button (needs a weak pull-down)
SW2	GP11	SW 2 Button (needs a weak pull-down)
SW3	GP12	SW 3 Button (needs a weak pull-down)
SW4	GP13	SW 4 Button (needs a weak pull-down)
SW5	GP22	SW 5 Button (needs a weak pull-down)
R1	A0 / GP26	Right potentiometer
R2	A1 / GP27	Left potentiometer
I_3V3	A2 / GP28	3.3V Current Sense resistor voltage
SDA	GP14	I2C channel 1 serial data
SCL	GP15	I2C channel 1 serial clock
CTS	GP6	UART clear to send (w/ level shifter)
RTS	GP7	UART request to send (w/ level shifter)
TX	GP8	UART channel 1 TX (w/ level shifter)
RT	GP9	UART channel 1 RX (w/ level shifter)
ServoA	GP0	PWM channel 0A
ServoB	GP1	PWM channel 0B
ServoC	GP2	PWM channel 1A
SQW	GP3	Square wave output from the DS3231 module



Secondary Side of the Accessory Board

Digital Inputs

In the first lab, GPIO pins were used as digital outputs to control a LED (Light Emitting Diode). GPIO pins can also be used as digital inputs, although they will need to be configured as such in the code. We will use the buttons on the expansion board as an example of a digital input.

Buttons

Buttons are a very common interface device for human input controls. The (“normally-open”) buttons on the expansion board have two metal contacts inside them. When the button is pressed, these two internal electrical contacts are forced to touch each other. When the button is released, the connection between these two contacts is broken. Other kinds of buttons are available, which do exactly the reverse of this (i.e. ”normally-closed” buttons) where a button press results in a connection being broken, instead of one being made.

Buttons are simple to use and easy to understand, but many of them can be tricky to use in a program. This is because they are mechanical devices and they are not perfect. The internal metal contact, inside the button, can bounce a few times before settling into the desired stable position. This can cause a software program to register multiple button presses, even for a single button press. Equally, the program might register a button press when a button is released. This is called button “bounce”, and we need to consider it when programming the Pico and use appropriate button “debouncing” techniques.

Code

First, connect your Pico to your computer (so that it says “Connected”) then open a new file in your IDE and type in the following code. This file is `digital-input.py` and it is provided in BS with the other accompanying files for this lab. Run the script, and you should see the green LED turn on when you press the button SW5, and turn off when you release the button.

```
# Use the Pin method from the machine software library
from machine import Pin

# Define the inputs and outputs and assign them to software objects
# First argument is a GPIO pin number, rather than a physical pin number
led1 = Pin(18, Pin.OUT)
sw5 = Pin(22, Pin.IN, Pin.PULL_DOWN)

while True:
# Control the LED output values, based on the received button input
        if sw5.value() == 1:
                led1.on()
        else:
                led1.off()
```

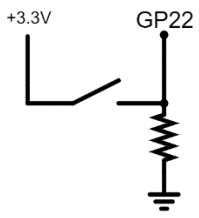
Code Analysis

Let's take a closer look at the code to understand what it does. The first line imports the Pin class from the machine module and we have seen this library module before. The next two lines create a Pin object called `led1` and set it to output mode, and a Pin object called `sw5` and set it to input mode with a large internal pull-down resistor.

Remember that a large resistor value (e.g. 50 kOhm) has a weak pulling effect on the voltage of an electrical node and vice versa. The effects of such a “weak” pullup resistor and can be overcome by the Pico’s output driver, once a low value is assigned to the output. However, when this output is undefined, the actual voltage line will “float” or default to a high value.

Pullup and pulldown resistors can be internal (i.e. like the ones that can be configured using software, inside an integrated semiconductor device like the Pico) or external (i.e. provided as discrete components, outside all integrated devices, like the pulldown resistors that are connected to the switches on the accessory board). More information about pull-up and pull-down resistors is given here (https://www.circuitbasics.com/pull-up-and-pull-down-resistors/).

The circuitry on the accessory board around button SW5 is equivalent to the following schematic (which is also given on the silkscreen of the board, for educational reasons only):



Most Pico GPIO pins are “digital” and so only measure voltages that are above or below certain threshold voltage values (i.e. near the midpoint voltage of +1.65V). Above this threshold, the pin will

be defined to be at a “high” logic level (or will have a program value of 1). Below this threshold, the pin will be defined to be at a “low” logic level (or will have a program value of 0).

When the button is not pressed, the input pin is connected to the ground through the pull-down resistor, so the GPIO pin voltage will be 0V (logic level of 0), since there are no voltage sources connected to it and pulling it high. When the button is pressed, the input pin is connected directly (or shorted) to the 3.3V power source, making the GPIO pin voltage 3.3V (i.e. a digital logic level of 1). Note that the weak pulldown to ground is still connected, but this should only have a marginal pull-down effect, provided that the resistance value to ground is large enough.

A pull-down resistor gets its name because it pulls the input pin down (i.e. in a negative direction) and keeps the input pin at a known “low” logic level, when the button is not pressed. Without the pull-down resistor, the input pin can float high or, worse, wander randomly between high and low values, during the times when the button is not pressed. The Pico also has configurable internal pull-up resistors, that would be configured using a `Pin.PULL_UP` command in our example and which can be used to achieve the same result as the external one used on the accessory board.

Q: Would a weak pull-down resistor have a large or a small value? What might happen to the voltage of a pin, if the pull-down value in the previous figure was really small (i.e. near to being zero Ohms or a direct short)? What else might happen?

A weak pull-down resistor would have a large value so it only gently pulls the pin to ground. If the resistor value was small, it would pull strongly to ground, making the pin always read 0V. Another thing that might happen would be a short circuit if another source tries to drive it high.

The last four lines of the previous code snippet execute inside a while loop, with a permanently `True` “condition”. This means that the instructions in the loop will be executed repeatedly, forever, until power is reset or new code is downloaded.

```
while True:
        if sw5.value() == 1:
                led1.on()
        else:
                led1.off()
```

Inside the loop, a conditional `if` statement is used to check if the button is pressed. If the statement followed by the `if` keyword is `True`, the code inside the `if` block will be executed, otherwise, the code inside the `else` block will be executed. If more inputs and outputs were available, more complicated intermediate `elif` blocks, with compound conditional statements, could also be used.

The digital value of the switch is read using the `sw5.value()` pin object method. This will return `1`, if the logic level on the pin is 1. Similarly, it will return `0` if the logic level on the pin is 0. You should now be able to determine which of these conditions corresponds to a button being pressed or released. This value is compared with the constant `1`, using the `==` operator, returning “`True`”, if the two

values are equal, and `False` otherwise. If `True`, the led is turned on, else it is turned off, using the Pin object `on()` or `off()` methods, respectively.

Coding Challenges

1. Modify the code to make the LED turn on when the button is not pressed, and turn off when the button is pressed (hint: Use `!=` to check for inequality, rather than inequality, or use a different method of your choice).
2. Utilize the other four buttons on the expansion board to control the other four onboard LEDs (plus the LED on the Pico, itself). See the Expansion Board section for more information.
3. Write a program `toggle.py` to make the LED toggle its state every time that you press a button and *then* release it. You will need to deal with the "button bounce" issue. See the previous Button section for more information on what this means. The LED should be off when the program starts. When the button is pressed, the LED should turn on. When the button is pressed again, the LED should turn off. Proper button debouncing means that the LED should only change its state once, for each button press.
    - How many ways can you think of to solve this problem (***hint***: there are at least two!)? Which one do you think is the best and why? What might affect your answer? Discuss this, briefly, in the space below.

I believe there are several ways to solve this problem with an infinite amount of ways of rewriting the code ;)

I believe method 2 is the best method because it is functional and it is only 5 lines of code.

Things that might affect my answer is if the code needs to do other things at the same time, like press other buttons (I tested this and it does not, only because the other buttons werent implemented directly in my while True loop).

The `toggle.py` program will need to be submitted for this lab in BrightSpace and also using github (see the later section for the instructions on how to do this). You are expected to solve the lab assignment on your own. However, you can ask for help if you are stuck. You should not simply copy large amounts of other people's code ("*Copy understanding, not answers!*"). You can use any LED and any button that you like on the expansion board.

Analog Inputs and PWM Outputs

The Pico has 3 Analog Digital Converter (ADC) input channels and 16 Pulse Width Modulation (PWM) output channels. The ADC channels can be used to read analog voltages, and the PWM channels can be used to generate PWM signals, which are actually digital outputs, controlled in a special way. Some microcontrollers have true analog outputs, but the Pico does not.

ADC

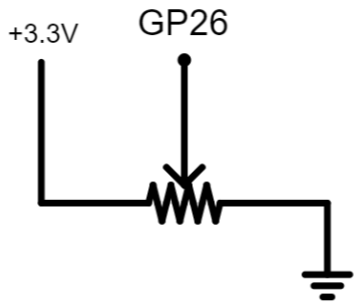
The ADC input on the Pico works by sampling the input voltage at a fixed rate and converting the samples to a digital value using a successive approximation method. The resolution of the ADC is 12 bits, which means it can represent 4096 different values. The input voltage range of the ADC is 0V to 3.3V, so each step or increment/decrement of the ADC will be 3.3V/4096 ~= 0.0008V. These inputs are designed only to read positive voltages only up to 3.3V. Negative enough voltages or high enough over-voltage conditions can damage the Pico.

Potentiometer

To create an analog voltage input, the accessory board uses a potentiometer, which is a three-terminal variable resistor with a sliding or rotating “wiper” contact that forms an adjustable voltage divider. The voltage between the middle terminal and one of the other terminals is a fraction of the input voltage determined by the ratio of the two resistors. The potentiometer can be used as a voltage divider by connecting the input voltage to the two outer terminals and the middle terminal to the ADC input pin.

The expansion board has two potentiometers. The middle terminal of the left potentiometer is connected to GPIO 26, and the middle terminal of the right potentiometer is connected to GPIO 27. The two other terminals are connected to the 3.3V power source and the ground. This means that the voltage between the middle terminal and the ground will vary between 0V and 3.3V, depending on the specific position of the potentiometer “wiper”.

The connection of the left potentiometer is equivalent to the following schematic:



Left Potentiometer Connection on the Accessory board

PWM

The PWM signals are generated by toggling the output pin at a fixed frequency. The duty cycle of the PWM signal can be adjusted to control the average voltage of the signal. In this lab, we are going to use the PWM channels to control the brightness of LEDs. You can learn more about PWM here (https://learn.sparkfun.com/tutorials/pulse-width-modulation/all).

However, as a summary, a PWM signal is periodic and repeats with a specific High/Low duty cycle that can be varied under program control. If the PWM output is mostly high, the average signal level will be higher than when it is mostly low. Many devices with digital inputs can be controlled to turn on and off using PWM signals. Dimmer switches for household lighting often work by varying the duty cycle of electricity to the light. For newer LED-based lights, a digital PWM signal is used.

Code

First, make sure your Pico is connected to your computer, then open a new file in your IDE and type in the following code (`adc-pwm.py` in the BS folder).

```
from machine import Pin, PWM, ADC
import time

adcA = ADC(Pin(26))
led1 = PWM(Pin(18))
led1.freq(1000)
# This could also be written as: led1 = PWM(Pin(18), freq=1000)

while True:
        value = adcA.read_u16()
        print(value)

        led1.duty_u16(value)

        time.sleep_ms(50)
```

Code Analysis

Looking at the code, the first line imports the `Pin`, `PWM`, and ADC classes from the `machine` module, PWM is used to control the PWM channels, and ADC is used to read the ADC channels. You can find more information about the `machine` module in the MicroPython Documentation (https://docs.micropython.org/en/latest/library/machine.html).

```
from machine import Pin, PWM, ADC
```

The third line creates an ADC object called `adcA`, and sets it to read from the pin GPIO 26, which is the left potentiometer on the accessory board.

```
adcA = ADC(Pin(26))
```

The next line creates a PWM object called `led1` and sets it to output mode, followed by a line to set the frequency of the PWM signal to 1000Hz. This means that the output pin will be toggled between high and low at a rate of 1000 times per second.

```
led1 = PWM(Pin(18))
led1.freq(1000)
```

This is going to have the effect of turning the LED on and off 1000 times per second. What do you think that the LED will look like, when this is happening? How fast a flicker can a human eye detect (hint: https://en.wikipedia.org/wiki/Flicker_fusion_threshold)?
I think the LED would just look like it's constantly on. We'd maybe see occasional flickers depending on the brightness.

We don't need to specify if a pin is an input or an output, when creating the PWM and the ADC objects. A PWM object can *only* be used to control an output pin, and an ADC object can *only* be used to read from an input pin. Next, we enter a while loop, as before. Inside the loop, we first read the value of the ADC using `adcA.read_u16()`, which returns a 16-bit unsigned integer ranging from 0 to 65535, This represents the latest raw measurement made by the ADC circuitry inside the Pico. We then print this value to the Shell window using `print(value)` command.

The duty cycle of the PWM signal is set to the same value, as just read from the ADC, using: `led1.duty_u16(value)`, which accepts a value between 0 and 65535. The duty cycle of a PWM signal is defined as the percentage of time that the signal is on, between the rising edges (or falling edges) of that signal. For example, if the duty cycle is 50%, the signal will be on for half of the period and off for the other half.

`led1.duty_u16(32768)` gives a 50% duty cycle, which results in an average voltage of the PWM output signal that is half the maximum value of 3.3V or about 1.65V. As already mentioned, a higher value means that the signal is high for a greater percentage of the time, but the output voltage is still digital, swinging between 0V and 3.3V at a rate of 1000 times a second. The `sleep` command then allows a human user enough time to read the value, before the next one is displayed.

```
while True:
        value = adcA.read_u16()
        print(value)

        led1.duty_u16(value)

        time.sleep_ms(50)
```

If the Pico only has a 12-bit ADC and the `read_u16` method has 16-bit granularity (i.e. varies from 0 to 65535). When mapping a 12-bit value into the upper bits of a 16-bit variable, how large is the resulting jump between each 16-bit value, assuming that the highest value is near the largest possible 16-bit value and the lowest 16-bit value is near zero?

Each step of the 12-bit ADC corresponds to a jump of about 16 in the 16-bit value.

If the frequency is set to 1000Hz, how long does the LED stay on, for each cycle, if the input to the duty_u16 function is set to a duty cycle of 42? What about a value of 65000? Which setting will appear brighter?

If the frequency is 1000Hz and duty is 42, the LED will be on for under 1% of the time and the brightness would be very low, whereas if duty was 65000 the LED would be on for over 99% of the time and the brightness would be very high.

How would you determine your own flicker fusion frequency threshold value (https://en.wikipedia.org/wiki/Flicker_fusion_threshold) using this equipment (you do not need to write the code for this, but just say what you'd do, in high-level terms?

To determine my own flicker fusion frequency threshold, I'd write a short program that gradually increases the LED's blink frequency using PWM. I'd start at a low frequency and slowly raise it while observing the LED. When the flicker disappears I'd record that frequency as my threshold.

Coding Challenges

1. What is the minimum and maximum value that you ever got back from the ADC?
    a. What do you expect the minimum and maximum value to be?
    b. Is there a difference?
    c. What could the reason be for that difference, if any?

Actual: 65535 and 304
Expected: 0 and 65535
Reasons: The Pico's ADC can't perfectly reach exactly 0 V or 3.3 V

---

2. Make the potentiometer control the blinking interval of the LED in the first code example.
3. Use PWM to make a "breathing" LED, i.e. the LED should gradually fade in and out, in terms of its brightness, periodically, as if it was breathing.
4. Make a progress bar using the four onboard LEDs. The progress bar should be controlled by the potentiometer to make progress start from zero (i.e. no LEDs lit) and go up to a maximum value (i.e. with all of the LEDs lit). Half of an LED is a half-bright LED.

Git

Git is a version control system that allows you to track changes to your code and collaborate with others. It is a very powerful tool that is widely used in the software industry. In this course, we will use git to manage our code. Git was originally authored by Linus Torvalds, the creator of Linux.

You can find the documentation for git at https://git-scm.com/docs. You can also find many tutorials and guides online. We will only cover the basics of git using a graphical user interface (GUI) in this lab. You are encouraged to learn more about git on your own. In future labs, we will learn more about git and its command line interface.

Repository

A git repository is a collection of files and folders that are tracked by git. You can think of a git repository as just a folder that contains all the files and folders in your project. You can create a git repository for each of your projects, and you can also create a git repository for your entire project folder.

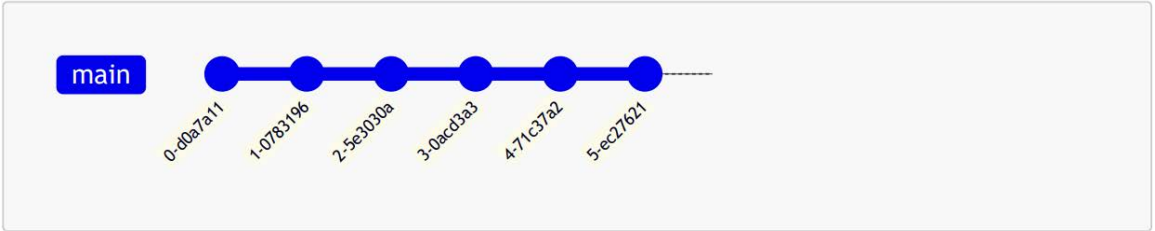
Version Control System

You can imagine git as a time machine for your code. You can use it to go back in time and see what your code looked like, at a certain point in time. You can also use it to create a new timeline for your code, so you can try out new ideas without worrying about breaking your code.

Commit Operations

Each commit takes a snapshot of your code at a certain point in time. You can go back to any commit and see what your code looked like at that time. You can also create a new commit to save the current state of your code.

Each commit has a unique identifier, called a hash, which is a long string of letters and numbers. You can use the hash value (which should each be unique) to refer to a specific commit. Usually, you only need to use the first 7 characters of the hash to refer to a commit, but keep in mind that the actual hash is much longer, and in the rare case where two commits have the same first 7 characters, you will need to use more characters to refer to the commit.

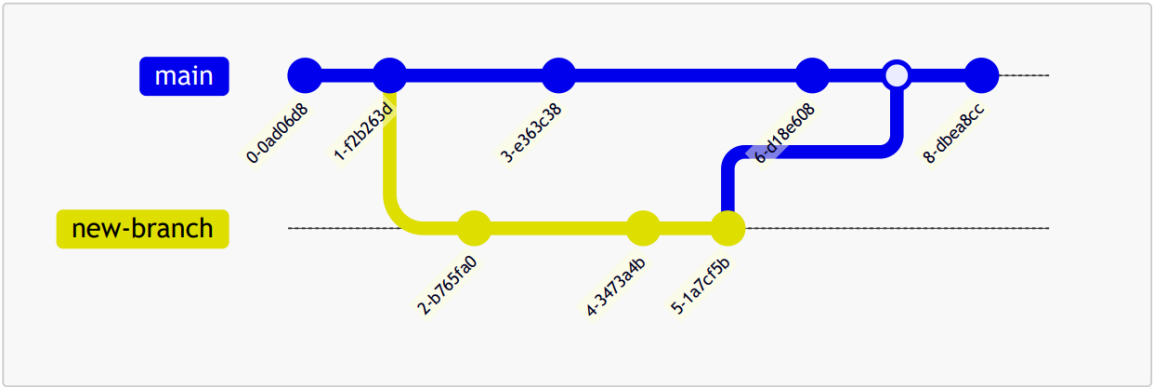


Each commit also has a commit message. This is a short message, when you create a commit, to describe what changes you have made to the code. You can also add a longer description if you want. You can use the commit message to describe what changes you have made to the code, and why you made those changes.

If you don't do this, you probably won't remember what code was in a particular commit, or what specific changes were made. Therefore, it is very important that you make your commit message a meaningful and understandable one. Otherwise, you (or other people) will find it a lot harder to look back in time and be able to tell what changes were made, without reviewing the entire codebase and all changes! Regular commits are worthwhile, each with a useful and specific commit message (i.e. ***not*** something generic like "changes made"!).

Branches

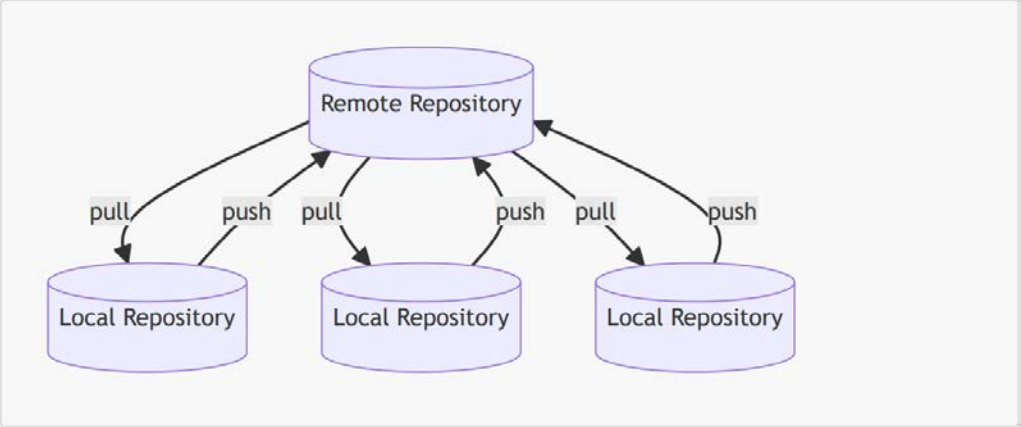
A branch is a timeline of commits. The main branch is usually called the master (or just the main) branch. You can create new branches to try out new ideas or when writing new features without affecting the main branch. You can then also "merge" a new branch back into the main branch, when you are done, so that others can see and use your changes.



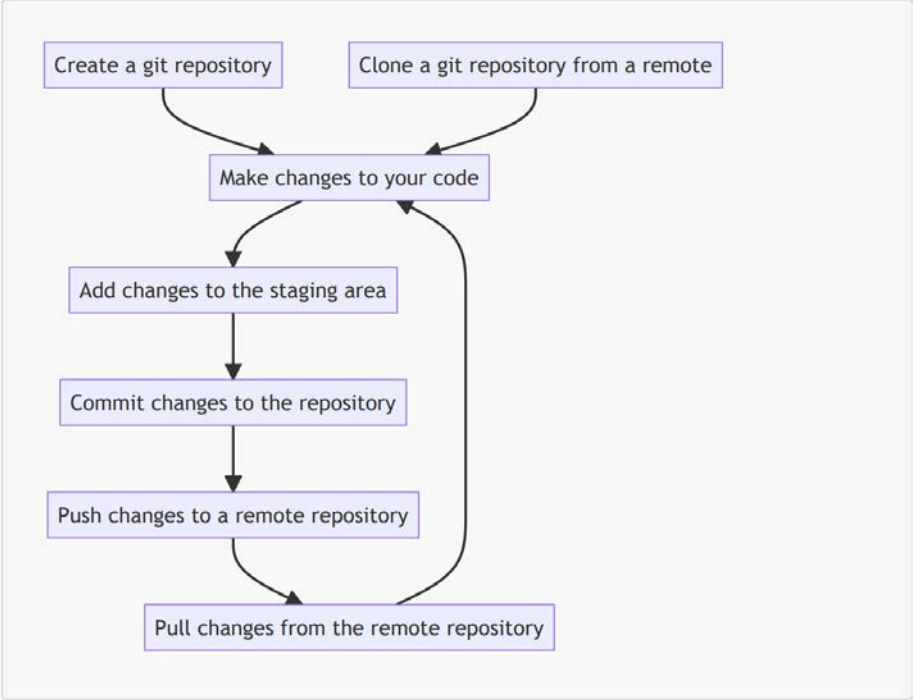
You can have multiple branches, and you can merge branches together. You can also delete branches when you no longer need them. When merging branches, git will try to automatically merge the changes from the two branches together. If there are conflicts, then you will need to manually resolve each of them. We will learn more about merging branches in future labs, but mostly conflicts arise when multiple people are making changes to the same thing simultaneously, so avoid this situation.

Distributed Version Control System

Git is a distributed version control system, which means that every developer can have a copy of the entire codebase. This way, you can work on your code, even when you are not connected to the Internet. You can also use git to back up your code to a remote server, so that you don't have to worry about losing your code, if your computer breaks.



To update your local repository with the changes from the remote repository, you "pull" the changes from the remote repository to your local repository. To update the remote repository with the changes from your local repository, you "push" the changes from your local repository back to the remote repository. The remote repository is usually hosted on a server, such as GitHub. However, you can also host your own remote repository on your own server. The process of using git to manage your code is as follows:



When you are working on your code, you make changes to your code, and you add the changes to the staging area. When you are ready, you commit the changes to the repository. You can then push the changes from the staging area to a remote repository, or pull changes that have been made by others from the remote repository, before starting your own work.

GitHub

GitHub is a website that allows you to host git repositories. You can use GitHub to collaborate with others and back up your code. GitHub is also like a social network for developers, you can follow other developers, and other developers can follow you. You can also see what other developers are working on, and the other developers can see what you are working on.

GitHub is the largest and most popular git hosting service, many open-source projects are hosted on GitHub. It is a good idea to create a GitHub account and get familiar with GitHub. You can use your GitHub account to showcase your projects and build your portfolio, which can help you get a job in the future.

Creating a GitHub Account

If you don't have a GitHub account already, go to https://github.com/ and click the *Sign up* button. You can use your uOttawa email address or your personal email address to create your GitHub account. If you decided to pursue a career in software development, it is a good idea to use your personal email address, in case you lose access to your uOttawa email address after you graduate. You can always change your email address later.

Create a Repository

After you have created your GitHub account and logged in, you should see your dashboard at https://github.com/. You should also see a green *New* button on the left side of the page. Click the *New* button to create a new repository.

This will take you to the *Create a new repository* page. Enter something like SED-1115-Lab-3 as the repository name. Keep the repository public, and don't check the *Initialize this repository with a README* option. Similarly, don't check the *Add .gitignore* option nor the *Choose a license* option.

Click the *Create repository* button, and you should see the repository page. It is empty because you have not pushed any code to the repository yet. You will learn how to push your code to GitHub in the next section.

GitHub Desktop

Git is a command-line tool, which means you need to type commands into a command window to use it. However, there are many graphical user interfaces (GUI) for git. These GUIs can make it easier to use git. In this lab, we will be using **GitHub Desktop.**

You can download GitHub Desktop from https://desktop.github.com/. Select the appropriate installer for your operating system, and follow the instructions to install GitHub Desktop. After you have installed GitHub Desktop, open it and sign in with your GitHub account.

Clone a Repository

After you have signed in, you should see an *Add* button on the left side of the window. Click the *Clone repository* button to clone a repository from GitHub. This is going to populate the code that you select (i.e. the one that you created earlier or others that you or others have created) into a local directory.

You should see a dialog with a list of your repositories, select the repository that you created earlier. Select the local path where you want to store the repository, and click the *Clone* button. You should now see the repository in the GitHub Desktop window. You can click the *Repository -> Show* in Explorer or *Repository -> Show* in the Finder button to open the repository folder in the File Explorer of your operating system.

The repository folder should be empty, except for a hidden `.git` folder, which contains the files and folders that git uses to track your code. You should not modify the files and folders in the `.git` folder, otherwise you might break git.

Add Files to the Repository

Now let's add some files to the repository. Try adding the code `toggle.py` from earlier to the repository folder. You can use the File Explorer of your operating system to copy and paste the file into the repository folder. If you haven't written the code yet, then you can create a new empty file in your IDE and save it as toggle.py in the repository folder. It is a good idea to use git to manage your code right from the beginning, so that you can easily go back to a previous version of your code if you make a mistake or if your computer dies, etc.

After you have added the file to the repository folder, you should see the file in the GitHub Desktop window. You can click the *Changes* tab to see the list of “staged” and “not staged” changes you have made to the repository. By default, GitHub Desktop automatically detects the changes that you have made to the repository and stages the changes for you.

Commit Changes

In the *Changes* tab, check the box next to the file that you want to commit, and enter a (**MEANINGFULL** and **USEFUL**) commit message in the *Summary* box. The commit message should be a short description of the changes that you have made. You can also enter a longer description in the Description box.

Once you are ready, click the *Commit to main* or *Commit to master* button to commit the changes to the main branch. This will create a new commit in the main branch, and you should see the commit in

the History tab too. You can click on the commit to see all of the changes that you have made in the commit.

Push Changes

After you have committed your changes, you can "push" the changes to the remote repository. Click the *Push origin* button to push the changes to the remote repository. As mentioned earlier, this will update the remote repository with the changes that you have made. You can now go to the repository page on GitHub to see these changes

Lab Submission:

Please submit the filled-in laboratory manual PDF file into BrightSpace, with a link to your github, and another copy of the `toggle.py` file into BrightSpace. The GitHub repository should contain the same `toggle.py` file and should be set to "*public*", so that the TA can access it. You can provide a functional link to this github in the space that is provided below:
https://github.com/Dxher/SED1115/tree/main/Lab3

The code should be well-formatted and well-commented. The code should be easy to read and understand.

Due Date:

See BrightSpace.

Additional Links and Resources

Interactive Pinout Diagram for Pico: https://pico.pinout.xyz/

Interactive Pinout Diagram for Pico W: https://picow.pinout.xyz/

Raspberry Pi Documentation:
https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html

Pico Datasheet: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf

Pico W Datasheet: https://datasheets.raspberrypi.com/picow/pico-w-datasheet.pdf

MicroPython Documentation: https://docs.micropython.org/en/latest/

Pull-Up and Pull-Down Resistors: https://www.circuitbasics.com/pull-up-and-pull-down-resistors/

Pulse Width Modulation: https://learn.sparkfun.com/tutorials/pulse-width-modulation/all

Git Documentation: https://git-scm.com/docs