CRICKIT WobblyBot
Created by Dave Astels

https://learn.adafruit.com/crickit-wobblybot

Last updated on 2021-11-15 07:21:33 PM EST
Table of Contents

Overview 3
• The body 3
• The Controllers 4
• Power Options 5
• Supplies and Tools 6

Building the Robot Chassis 7
• The Platform 7
• Leg Motors 8
• The Upper Deck 9
• Legs 10
• Mounting the CRICKIT and Controller 13
• Wiring 14
• Adding a Wagging Tail 14

Power 18

Circuit Playground Express and CircuitPython 22

Feather M0 Bluefruit and Arduino 27

Micro:bit and MakeCode 40

Wrapup 44
Overview

In this guide we will build a cute little robot body all employing the principles of hopping instead of rolling. ([https://adafruit.com/product/2943](https://adafruit.com/product/2943)) We'll even do it with 3 different brains!

- Circuit Playground Express with code in CircuitPython that can be controlled programatically,
- Feather M0 Bluefruit with code in Arduino/C that can be controlled programatically but also via Bluetooth, and
- BBC micro:bit with code in MakeCode that can also be controlled programatically.

In each case, the hardware will be the same. The controller board, matching CRICKIT, and code will vary. Part of the reason is to show how all the CRICKIT variations do the same job and the choice is based on the controller board and language you want to use.

The body

1 x Anodized Aluminum Metal Chasis

This Anodized Aluminum Metal Chasis for a Mini Robot Rover does exactly what it says. You can build a very sturdy (and quite handsome!) little robot rover with this metal frame.
1 x **Top Metal Plate for a Mini Robot Rover Chassis**
You don't need it but it let's you add a little more storage to your bot. Perfect for housing your robot's electronics and batteries underneath. This plate comes with two standoffs and the appropriate screws.

https://www.adafruit.com/product/2944

4 x **Continuous Rotation Micro Servo - FS90R**
This little micro servo rotates 360 degrees fully forward or backwards, instead of moving to a single position. You can use any servo code, hardware or library to control these servos. Good for making simple moving robots. Comes with five horns and attachment screw.

https://www.adafruit.com/product/2442

1 x **Black Nylon Screw and Stand-off Set – M2.5 Thread**
Or 4 pairs of other M2.5 x 6mm nuts and bolts

https://www.adafruit.com/product/3299

1 x **Mini Geared Pager Motor with Return Spring**
At the core is a 'pager motor' - a small thin DC motor that runs on about 3VDC. That motor is then geared down inside the red-plastic body of the motor. Finally, the output axle is connected to a plastic triangle that has a bent-wire return spring.

https://www.adafruit.com/product/3871

---

**The Controllers**

Pick one or more controller board and CRICKIT combinations.

1 x **Circuit Playground Express**
Circuit Playground Express is the next step towards a perfect introduction to electronics and programming. We've taken the original Circuit Playground Classic and made it even better! Not only did we pack even more sensors in, we also made it even easier to program.

https://www.adafruit.com/product/3333

1 x **Adafruit CRICKIT for Circuit Playground Express**
That's our Creative Robotics & Interactive Construction Kit. It's an add-on to our popular Circuit Playground Express that lets you #MakeRobotFriend using CircuitPython, MakeCode (coming soon), or Arduino.

https://www.adafruit.com/product/3093
1 x Adafruit Feather M0 Bluefruit LE
This is the Adafruit Feather M0 Bluefruit LE - our take on an 'all-in-one' Arduino-compatible + Bluetooth Low Energy with built in USB and battery charging. It's an Adafruit Feather M0 with a BTLE module, ready to rock!

1 x Adafruit CRICKIT FeatherWing for any Feather
Plug in any Feather mainboard you want into the center, and you're good to go! The Crickit is powered by seesaw, our I2C-to-whatever bridge firmware. So you only need to use two I2C data pins to control the huge number of inputs and outputs on the Crickit. All those timers, PWMs, sensors are offloaded to the co-processor.

1 x BBC micro:bit
Designed specifically for kids and beginners, the micro:bit is a pocket-sized computer that you can code, customize and control to bring your digital ideas, games and apps to life. It’s a small, code-able device that is a non-intimidating introduction to programming and making – switch on, program it to do something fun – wear it, customize it, develop new ideas.

1 x Adafruit CRICKIT for micro:bit
Plug your :bit into the 40 pin edge connector and start controlling motors, servos, solenoids. You also get signal pins, capacitive touch sensors, a NeoPixel driver and amplified speaker output. It complements & extends micro:bit so you can still use all the goodies on the :bit, but now you have a robotics playground as well.

Power Options

1 x 5V 2A (2000mA) switching power supply - UL Listed
Use this at your bench when building and programming to save the batteries.

1 x 3 x AA Battery Holder with 2.1mm Plug
A holder for three (3) AA batteries! It's got an 8" long power cable with a 2.1mm DC jack at the end. An alternative to the large LiPo.
1 x Lithium Ion Battery Pack - 3.7V 6600mAh
Need a massive battery for your project? This lithium ion pack is made of 3 balanced 2200mAh cells for a total of 6600mA capacity! This is my go-to battery back for untethered CRICKIT projects, along with the boost board listed below.


1 x PowerBoost 1000 Charger - Rechargeable 5V Lipo USB Boost @ 1A - 1000C
PowerBoost 1000C is the perfect power supply for your portable project! With a built-in load-sharing battery charger circuit, you'll be able to keep your power-hungry project running even while recharging the battery! This little DC/DC boost converter module can be powered by any 3.7V LiIon/LiPoly battery, and convert the battery output to 5.2V DC for running your 5V projects.

https://www.adafruit.com/product/2465

1 x Lithium Ion Polymer Battery - 3.7v 1200mAh
Used to power the controller board.

https://www.adafruit.com/product/258

1 x Adafruit Mini Lipo w/Mini-B USB Jack - USB LiIon/LiPoly charger
You'll need this or something comparable to charge the LiPo unless you use Feather which has a changer buildin.

https://www.adafruit.com/product/1905

Supplies and Tools

- Corrugated cardboard
- Craft knife, cardboard saw, scissors, etc. to cut and shape cardboard
- hot glue and glue gun
- solder and soldering iron
- a couple pins worth of male header strip
- about 10cm (4 inches) of something stiff but not rigid. a piece of bungee-cord works well
- small zip ties to keep wiring neat and out of the way is useful
Building the Robot Chassis

The Platform

The chassis we're using is precut with various slots and holes to allow mounting things to it. It's a fairly small chassis so we'll also use the optional raised platform to provide a bit more space. The CRICKIT and controller will go on that.
Leg Motors

The chassis provides precut mounting holes on the sides for four microservo sized motors. However normal servos won't work for what this project needs, which is continuous turning motors. While there are DC motors in a microservo bodies, the CRICKIT can only drive 2. This design uses 4. Fortunately, continuous rotation microservos are available.

The servo mounting holes and the microservos accept M2.5 bolts. Nylon bolts worked nicely, are cheap and lightweight.

Mount the servos so that all four have their output shafts nearest the ends of the chassis. It should look like the third photo.
The Upper Deck

The chassis being used has a optional (available separately [https://adafruit.it/CBM](https://adafruit.it/CBM)) upper deck that comes in handy if you want to use a large battery pack (which is advised for motors). We'll put the CRICKIT and controller board on this platform with the battery below.
Legs
There was a fair bit of thought given to how this robot should move. The CRICKIT turtle project (https://adafruit.it/CBd) provided some inspiration. In the end a short leg was mounted on each servo. The servos come with a variety of horns, including a round disk. That provided the most surface area and support for the type of leg being used. One is mounted on each servo and secured with the included screw.

Legs were cut from corrugated cardboard with the corrugations running the length of the leg. 50mm (2in) long and 25mm (1 in) wide fit well. Each end was rounded using one of the servo disk horns to mark the curve. The legs will be short enough so that they don’t touch if they happen to point inward at the same time.

The cardboard legs are then hot glued onto the disks. If your environment has a lot of slippery floors (cushion flooring, hardwood, laminate, tile, etc) you may want to add some hot glue along the ends of the legs so that they'll grip a bit better.

Don't worry about the orientation of the legs; these are continuous rotation servos, so there's no limits or zero positions. Additionally, each motor will vary a bit so they won't always turn at the same rate. Far from being a problem, this gives the robot its quirky gait.
Mounting the CRICKIT and Controller

While you can mount a CRICKIT with double-sided foam tape, if you have a 3D printer (or can get something printed) it is really useful to make a CRICKIT mounting base. This is the one by the Ruiz brothers of 3D Hangout fame (https:// adafruit.it/BXU) which has some standout features:

- mounting slots
- notches for the cap-touch sensors, USB, and power
- various snap-in centers
- support for press/heat insertable threaded M3 inserts

For this project we use the insert with two mounting holes. These holes are the right distance apart to mount on the upper deck of the chassis: one at each end of the center slot. However, with a bolt through each mounting hole, the base sits slightly on top of the bolts securing the deck to its standoffs. This can be solved by cutting, grinding, or melting a slight notch where the bolt heads touch the mount. See the first picture. Depending on the M3 bolts you use to connect the mount to the deck, you might have to widen the holes slightly. See the second picture.

The final picture shows the mount, secured to the deck, with heat insertable M3 inserts in place. Now it's easy and mess free to swap out different CRICKIT/controller combos.
Wiring

The servo wiring has to be secured; there's enough slack on the two rear ones that it'll cause a problem if they're left free to get caught up in the legs.

Feed the servo wiring up through the slot at the back of the chassis. There is a wider section in the middle that makes it easy to get the connectors through. Connect them to the servo header on the CRICKIT:

1. front right
2. front left
3. rear right
4. rear left

Take up the slack, leaving enough to disconnect/reconnect the servos at the CRICKIT header. Bunch it together and secure using a couple small zip ties underneath the chassis. The holes/slots in the chassis can be used handily for this. On the back of the robot use a couple more zip ties to bunch the servo wiring together.

Adding a Wagging Tail

When a friend saw a video of an early version of the bot, they commented that it could use a wagging tail. Think about making a wagging tail for a moment. You need it to move back and forth through a relatively small angle: +/- 30 deg from center (i.e. straight up) is probably fine. Sounds like the ideal job for a servo. Except for a couple things. Even a micro servo is not really all that small, but a sub-micro servo might do the job.
Sub-micro Servo - SG51R
This is just about the cutest, tiniest little micro servo we could find, even smaller than the 9-gram micro servos we love so much. It can rotate approximately 180 degrees (90 in...
https://www.adafruit.com/product/2201

The other problem is that the Crickit has four servo drives, and all four are in use for the locomotion motors. It could be wired directly to the controller board. That's kind of hacky (advice: never let that stop you) but more importantly, part of the design goal is to make use of the CRICKIT.

There are two DC motor drives that aren't being used. If only we had a tiny DC motor that could wave something between +/- 30 degrees...

It turns out that there's a part that's perfect for doing that. It's small and moves an arm through about +/- 60 degrees. It's not especially powerful, but it doesn't have to be.

Mini Geared Pager Motor with Return Spring
These are very unusual motors, but we thought they could be handy for projects that need a small geared-down motor. At the core is a 'pager motor' - a small thin DC motor that...
https://www.adafruit.com/product/3871
The wires on this motor are very fine. Fine enough that they aren't easily used with the screw terminals of the motor drive. To fix this, we will solder them to a 2-pin piece of standard 0.1" header (https://adafruit.com/91). The pins are just right to fit into the motor connector. As usual, some heat shrink is handy to prevent shorts and to act as a strain relief.

The first attempt at mounting the tail motor involved using hot glue. With the regular back and forth motion of the tail, this didn't hold very well. Hot glue has its limits. In fact, glue in any situation that is under regular fluctuating stress might not be a great idea. Also, the plastic of the motor casing is quite thin, and apparently has a low melting point. Some deformity was noticed when it was eventually removed.

What did work well are zip ties. Specifically, four small ones. As shown in the pictures, two were placed around the back edge of the chassis using the slot that the servo wiring is routed through. These are then used as mount points for two more zip ties as shown in the photos. These two are secured around the bottom and back of the motor. This places the motor at an angle so that the rotation axis isn't horizontal. That's a good thing in that it angles the tail at about 45 degrees so it is completely clear of the CRICKIT's power plug.

The tail was made from a piece of heavy bungee cord. This had enough wave slightly, but is stiff enough not to
Power

Power for this project went through a few iterations.

During development the CRICKIT (and through it, the control board) was powered by a 5V 2A wall wart.

5V 2A (2000mA) switching power supply - UL Listed
This is an FCC/CE certified and UL listed power supply. Need a lot of 5V power? This switching supply gives a clean regulated 5V output at up to 2000mA. 110 or 240 input, so it works...
https://www.adafruit.com/product/276

This worked fine. When it came time to test it out untethered, a 3xAA battery pack was used. As before, this plugged into the CRICKIT and powered everything. This worked fine until the tail was added. At that point, the first time power was applied to the tail motor, the CircuitPython runtime crashed. In the course of investigating the problem it was found that things worked fine when the controller was connected to USB (for debugging), but crashed as before then battery powered. It was then that an oscilloscope was used to look at the power signal (sampling it using the ground and 5v connections on the NeoPixel connector). Here's a trace of the power when no motors are running.
There's a little noise, but it looks random and very low amplitude. Now look at the same thing with all four servos running at half speed, and the tail wagging.

That's some significant variation of the voltage! And that's not noise either. The frequent spikes are from the servos. Remember that at half speed we should be seeing a roughly 50% duty cycle wave from the PWMs that drive the servos. This looks about right. The less frequent drops in the signal are when the tail motor was powered. When you see the code you'll see that a burst of power is applied to the tail motor and turned off, allowing the spring to bring it back to center.
This is a problem. Motors don't really care so much about how clean their power is, but microcontrollers do. The noise from the PWMs don't seem to present a problem, but when the the tail motor kicks in that drop is enough to scramble the MCU.

The upshot of this is that the running several motors and the controller on the same power supply, especially if they are batteries, is going to be trouble. However, all of the controller boards we're using can be powered by a LiPo battery. Giving the controller it's own supply alleviates this problem.

```
The power supply circuit on the micro:bit is minimal and there's not a good way to avoid it being powered from the CRICKIT (through the 3.3v edge connector pad). Fortunately, the micro:bit (or the regulation on the micro:bit CRICKIT) is such that it's proven to be more resistant to the power supply fluctuations and hasn't suffered from being powered from the CRICKIT.
```

So, that will fix the problem from the fluctuations due to the motors. There's still the SAMD21 that runs the CRICKIT, and there's no way to supply it with a separate battery (or the battery from the controller board). In practice it doesn't seem as sensitive as the controllers. Just keep an eye on the charge on your battery; if it gets too low you'll have problems. Alkaline AAs seems to be especially prone to causing problems. Then again, this project runs five motors constantly which is quite demanding.
The final build went with a 6600 mAh LiPo and a 5v boost converter to power the CRICKIT and a 1200mAh LiPo for the controller board. Note that on the Circuit Playground Express CRICKIT, the Vout brass standoff was not used since it is being powered separately. However the ground connection must be maintained.

For safety and easier mounting there are files on Thingiverse (https://adafru.it/CBe) for cases for the 6600 mAh LiPo and the boost converter. Using these will avoid any chance of shorts on the bottom of the boost board, as well as protect the LiPo from accidental damage.

The LiPo (in its case) is mounted under the rack, not quite all the way back. It's fairly heavy and having it too far back makes the robot prone to going up on its hind legs.

Cute, but not very effective for moving. The boost converter was mounted on the nose of the bot for east access. The converter case was modified to allow use of a screw-terminal block instead of the USB-A connector. If you are using a AA battery holder, it can be placed under the rack in a similar manner.

There is room between them for the controller's LiPo.
We'll be using CircuitPython for this project. Are you new to using CircuitPython? No worries, [there is a full getting started guide here](https://adafru.it/cpy-welcome).

Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. [You can learn about Mu and its installation in this tutorial](https://adafru.it/ANO).

For the CircuitPython version the Circuit Playground Express was chosen for variety, although any of the M0 or M4 boards could be used.
It's advised that you use the Circuit Playground Express version of CircuitPython that includes the CRICKIT library. See this part of the CRICKIT guide for details and instructions (https://adafru.it/Bwa).

To start, servos are allocated and lists of them are made for later use.

```python
# Each servo corresponds to one of the legs
front_right = crickit.continuous_servo_1
front_left = crickit.continuous_servo_2
rear_right = crickit.continuous_servo_3
rear_left = crickit.continuous_servo_4

# Useful groups of legs
all_legs = [front_right, front_left, rear_right, rear_left]
front_legs = [front_right, front_left]
rear_legs = [rear_right, rear_left]
right_legs = [front_right, rear_right]
left_legs = [front_left, rear_left]
```

If you noticed when we mounted the motors, they weren't all oriented the same way. Look at the labels... on two you can see them, on the others you can't. The result of this is that to have all legs work to move forward, two motors have to run in the opposite direction to the others. We can make that adjustment in the code closest to the servos so that most of the code can ignore this detail. This mapping helps implement that by providing a multiplier (+/-1) for the speed of each motor. It allows higher level code to set a motor throttle with positive being forward without regard to the orientation of the servo.

```python
# The sign (+1/-1) for forward motion for each servo
direction_values = {front_right: +1,
                   front_left: -1,
                   rear_right: +1,
                   rear_left: -1}
```

The final bit of support data is value of the PWM limits for each servo. This can be adjusted for your servos so that setting the throttle to 0 causes them to stop.

```python
pwm_ranges = {front_right: (500, 2400),
              front_left: (500, 2400),
              rear_right: (500, 2400),
              rear_left: (500, 2400)}
```

We have a function to set the servo limits, and stop each of them.

```python
def init():
    for leg in all_legs:
        limits = pwm_ranges[leg]
        leg.set_pulse_width_range(min_pulse=limits[0], max_pulse=limits[1])
        leg.throttle = 0
```
The core of the movement code is in the `forward`, `reverse`, and `stop` functions. All three take a single leg or a list of legs. The `forward` and `reverse` functions also take a speed.

Notice how the `direction_values` list is used, and also how reverse multiplies the adjusted speed by -1 so that you can ask it to reverse with a positive speed.

```python
def forward(servo_or_servos, speed):
    if type(servo_or_servos) == list:
        for servo in servo_or_servos:
            servo.throttle = speed * direction_values[servo]
    else:
        servo_or_servos.throttle = speed * direction_values[servo_or_servos]

def reverse(servo_or_servos, speed):
    if type(servo_or_servos) == list:
        for servo in servo_or_servos:
            servo.throttle = speed * -1 * direction_values[servo]
    else:
        servo_or_servos.throttle = speed * -1 * direction_values[servo_or_servos]

def stop(servo_or_servos):
    if type(servo_or_servos) == list:
        for servo in servo_or_servos:
            servo.throttle = 0
    else:
        servo_or_servos.throttle = 0
```

Four other functions provide higher level behaviors that are useful in scripting motion.

```python
def rotate_clockwise(speed):
    forward(left_legs, speed)
    reverse(right_legs, speed)

def rotate_counterclockwise(speed):
    forward(right_legs, speed)
    reverse(left_legs, speed)

def crawl_forward(speed):
    forward(all_legs, speed)

def crawl_backward(speed):
    reverse(all_legs, speed)
```

Now for the tail. `wag_for` is the main function that is used, the others just provide the required abstractions. It wags the tail for the specified time. `Wag` is the function that makes the tail do its thing: power the tail motor for 100 ms, then wait for 250 ms while the spring recenters the tail.
def wag(speed):
    tail.throttle = speed
    time.sleep(0.1)
    tail.throttle = 0.0
    time.sleep(0.25)

def wag_for(seconds):
    target_time = time.monotonic() + seconds
    wag_throttle = 1.0
    while time.monotonic() < target_time:
        wag(wag_throttle)
        wag_throttle *= -1.0

Finally there's a sample script that runs forward for 5 seconds, rotates clockwise, run backwards for 2 seconds, rotate the other way and run forward again, finally stopping.

def demo1():
    crawl_forward(0.5)
    wag_for(5.0)
    rotate_clockwise(0.25)
    wag_for(3.0)
    crawl_backward(0.5)
    wag_for(2.0)
    rotate_counterclockwise(0.25)
    wag_for(3.0)
    crawl_forward(0.5)
    wag_for(5.0)
    stop(all_legs)
direction_values = {front_right: +1, 
    front_left: -1, 
    rear_right: +1, 
    rear_left: -1}

# Tweak the pwm ranges for each servo so that throttle of 0 stops the motor
pwm_ranges = {front_right: (500, 2400), 
    front_left: (500, 2400), 
    rear_right: (500, 2400), 
    rear_left: (500, 2400)}

def init():
    for leg in all_legs:
        limits = pwm_ranges[leg]
        leg.set_pulse_width_range(min_pulse=limits[0], max_pulse=limits[1])
        leg.throttle = 0

def wag(speed):
    tail.throttle = speed
    time.sleep(0.1)
    tail.throttle = 0.0
    time.sleep(0.25)

def wag_for(seconds):
    target_time = time.monotonic() + seconds
    wag_throttle = 1.0
    while time.monotonic() < target_time:
        wag(wag_throttle)
        wag_throttle *= -1

def forward(servo_or_servos, speed):
    if isinstance(servo_or_servos, list):
        for servo in servo_or_servos:
            servo.throttle = speed * direction_values[servo]
    else:
        servo_or_servos.throttle = speed * direction_values[servo_or_servos]

def reverse(servo_or_servos, speed):
    if isinstance(servo_or_servos, list):
        for servo in servo_or_servos:
            servo.throttle = speed * -1 * direction_values[servo]
    else:
        servo_or_servos.throttle = speed * -1 * direction_values[servo_or_servos]

def stop(servo_or_servos):
    if isinstance(servo_or_servos, list):
        for servo in servo_or_servos:
            servo.throttle = 0
    else:
        servo_or_servos.throttle = 0

def rotate_clockwise(speed):
    forward(left_legs, speed)
    reverse(right_legs, speed)

def rotate_counterclockwise(speed):
    forward(right_legs, speed)
    reverse(left_legs, speed)

def crawl_forward(speed):
forward(all_legs, speed)

def crawl_backward(speed):
    reverse(all_legs, speed)

def turtle():
    stop([rear_right, rear_left])
    stop(rear_left)
    forward(front_right, 0.5)
    forward(front_left, 0.5)

def snake_step():
    stop(all_legs)
    forward(right_legs, 0.5)
    time.sleep(1.0)
    stop(right_legs)
    forward(left_legs, 0.5)
    time.sleep(1.0)
    stop(left_legs)

init()

def demo1():
    crawl_forward(0.5)
    wag_for(5.0)
    rotate_clockwise(0.25)
    wag_for(3.0)
    crawl_backward(0.5)
    wag_for(2.0)
    rotate_counterclockwise(0.25)
    wag_for(3.0)
    crawl_forward(0.5)
    wag_for(5.0)
    stop(all_legs)

demo1()
This build will use the FeatherWing CRICKIT. The Feather you choose will determine your language options as well as any non-CRICKIT capabilities. Here we'll use the Feather M0 Bluefruit and write the code in C/C++ using the Arduino framework and tools. This board was selected for its bluetooth capabilities which will let us drive the robot from the Adafruit Bluefruit app controller interface, as well as run pre-written sequences using the controller's 1-4 buttons.

You'll need to have the Arduino IDE installed as well as the appropriate board packages and libraries. The Feather M0 Bluefruit LE guide (https://adafruit.it/CBf) covers this in detail. It's a good idea to read through that guide (https://adafruit.it/CBf) if you haven't yet; it will show you all the tricks of this board.

This code has a couple support files with it that were copied from the controller.ino example: BluefruitConfig.h and packetParser.cpp. Either start a new sketch called WobblyBot and copy them into the directory along with the WobblyBot.ino file below. Or clone the repo from github; it has everything in place that's needed. In either case, load WobblyBot into the Arduino IDE, set your board and port (see the linked guide for this Feather board), and compile/upload the code.

Use of this version of the CRICKIT is identical to the others. Connect the servos to the appropriate connectors on the CRICKIT, and connect the tail motor to the motor 1 connections. The smaller LiPo powers the Feather board. Using a Feather has the advantage that the battery doesn't need to be disconnected to charge it. It will be charging whenever the Feather is connected via USB. Finally connect the 5v supply to the CRICKIT.

We won't go into detail on the boilerplate BLE setup and use. That's covered in the linked guide and the comments from the example code that was used have been kept intact.

While the structure of the C++ code is similar to the Python, the details are, naturally, quite different.

We start by creating servo objects, placing them in an array. We define a constant index for each leg. We have the multipliers to correct for servo orientation, which servo pin to use for each leg, and the limit settings to tweak. Additionally there is an array of strings used in debug output for leg names.

```cpp
Adafruit_Crickit crickit;
seesaw_Servo legs[] = {seesaw_Servo(&crickit),
seesaw_Servo(&crickit),
seesaw_Servo(&crickit),
seesaw_Servo(&crickit)};
```
const int front_right = 0;
const int front_left = 1;
const int rear_right = 2;
const int rear_left = 3;

// Left and right motors turn in the opposite direction
const float motor_directions[4] = {+1.0, -1.0, +1.0, -1.0};

// pins to connect each servo to
const int servo_pins[4] = {CRICKIT_SERVO1,
                            CRICKIT_SERVO2,
                            CRICKIT_SERVO3,
                            CRICKIT_SERVO4};

// PWM ranges for each motor, tune these so that setting the angle to 90 stops the motor
int pwm_ranges[4][2] = {{500, 2400},
                        {500, 2400},
                        {500, 2400},
                        {500, 2400}};

const __FlashStringHelper *leg_names[4] = {F("Front right"),
                                           F("Front left"),
                                           F("Rear right"),
                                           F("Rear left")};

Speaking of debugging, there are two small functions for outputting errors and information. If you want to output to the serial console simple uncomment the #define DEBUG line near the start of the file. Remember to recompile/upload with it commented out before running the robot untethered.

void error(const __FlashStringHelper *err)
{
    digitalWrite(13, HIGH);
    #ifdef DEBUG
    Serial.println(err);
    #endif
    while (1);
}

void log(const __FlashStringHelper *msg)
{
    #ifdef DEBUG
    Serial.println(msg);
    #endif
}

Using the seesaw_servo library lets us set servo angles, but there is no direct support for setting speed as in the Python library, we need to set the raw angle. To abstract this away, there's the function speed_to_angle. It takes a floating point speed from -1.0 to 1.0 and converts it to an integer angle from 0 to 180.

To hide the details, and to provide a single place to log servo settings, we have the set_leg function.
```c
int speed_to_angle(float speed)
{
    return (int)(speed * 90.0 + 90.0);
}

void set_leg(int leg, float speed)
{
    int angle = speed_to_angle(speed * motor_directions[leg]);
    #ifdef DEBUG
    Serial.print(F("Setting "));
    Serial.print(leg_names[leg]);
    Serial.print(F(" to "));
    Serial.println(angle);
    #endif
    legs[leg].write(angle);
}
```

Now we have the forward/reverse/stop functions. This is quite different than Python. In the Python version we could pass a single leg or a list of legs and the function looked at what it was given and did the right thing. C++ does not have that capability due to its very different approach to typing. It does, however, have a way to send a variable number of arguments to a function. We'll use that here. As an example, consider the **stop** function:

```c
// Stop the listed motors
// -1 required as the last argument

void stop(int leg, ...)
{
    va_list args;
    va_start(args, leg);
    log(F("Stop"));
    while (leg != -1) {
        set_leg(leg, 0.0);
        leg = va_arg(args, int);
    }
    va_end(args);
}

void stop_all()
{
    stop(front_right, front_left, rear_right, rear_left, -1);
}
```

We are using the variable argument support provided by the **va_*** macros/functions. This starts in the function's signature: the **...** in the parameter list tells the compiler that this function can have an arbitrary number of arguments (but at least one: **leg**). The first thing the function does is create a variable of type **va_list**. This will hold the bookkeeping information for accessing the arguments. That variable gets initialized using **va_start**, passing it the **va_list** and the name of the argument immediately before the **...** in the parameter list.
As it traverses through the arguments, note that it doesn’t have names to use to get the values of any past the first one. Instead `va_arg` is used. This is passed the `va_list` that was initialized earlier as well as the type of the next expected argument. In our case they are all leg indexes so are all `int`.

Notice the comment saying that the last argument has to be -1. -1 is not a valid leg index (they are 0-4) so we are using that to mark the end of the arguments. The while loop continues to set leg speeds to 0 and fetching the next leg index until a -1 is found. This is called a sentinel value. Its use is solely to be the final argument so the loop knows when to stop.

The `stop_all` function shows a use of this.

The `forward` and `reverse` functions work in the same way, except that a speed value is the initial argument.

```c
void forward(float speed, ...) {
    va_list args;
    va_start(args, speed);
    int leg = va_arg(args, int);
    log(F("Forward"));
    while (leg != -1) {
        set_leg(leg, speed * motor_directions[leg]);
        leg = va_arg(args, int);
    }
    va_end(args);
}

void forward_all(float speed) {
    forward(speed, front_right, front_left, rear_right, rear_left, -1);
}

void reverse(float speed, ...) {
    va_list args;
    va_start(args, speed);
    int leg = va_arg(args, int);
    log(F("Reverse"));
    while (leg != -1) {
        set_leg(leg, speed * -1 * motor_directions[leg]);
        leg = va_arg(args, int);
    }
    va_end(args);
}

void reverse_all(float speed) {
    reverse(speed, front_right, front_left, rear_right, rear_left, -1);
}
```
The rotation functions use the forward and reverse functions as well, this time with 2 legs each.

```c
void rotate_clockwise(float speed) {
  forward(speed, front_left, rear_left, -1);
  reverse(speed, front_right, rear_right, -1);
}

void rotate_counterclockwise(float speed) {
  forward(speed, front_right, rear_right, -1);
  reverse(speed, front_left, rear_left, -1);
}
```

The tail works the same way as in the Python code. The only real difference is that the wagging is handled in the main loop rather than in a timed loop because we want to check for commands frequently.

```c
seesaw_Motor tail(&crickit);
boolean tail_power = 0.5;

//...
void wag(float speed) {
  tail.throttle(speed);
  delay(75);
  tail.throttle(0.0);
  delay(50);
}
//...
void loop() {
  wag(tail_power);
  tail_power *= -1.0;
  //...
}
```

The rest of the main loop checks for a command via BLE and performs the appropriate action based on which button was pressed (or released).

```c
void loop() {
  // wag(tail_power);
  tail_power *= -1.0;

  // Wait for new data to arrive
  uint8_t len = readPacket(&ble, BLE_READPACKET_TIMEOUT);
  if (len == 0) return;

  // Got a packet!
  // printHex(packetbuffer, len);

  // Buttons
  if (packetbuffer[1] == 'B') {
```
```c
uint8_t buttnum = packetbuffer[2] - '0';
boolean pressed = packetbuffer[3] - '0';

#ifdef DEBUG
    Serial.print("Button "); Serial.print(buttnum);
    if (pressed) {
        Serial.println(" pressed");
    } else {
        Serial.println(" released");
    }
#endif

switch(buttnum) {
    case 1:
        if (pressed) {
            demo1();
        }
        break;
    case 2:
        if (pressed) {
            demo2();
        }
        break;
    case 3:
        if (pressed) {
            demo3();
        }
        break;
    case 4:
        if (pressed) {
            demo4();
        }
        break;
    case 5:
        if (pressed) {
            rotate_counterclockwise(0.5);
        } else {
            stop_all();
        }
        break;
    case 6:
        if (pressed) {
            rotate_clockwise(0.5);
        } else {
            stop_all();
        }
        break;
    case 7:
        if (pressed) {
            reverse_all(0.5);
        } else {
            stop_all();
        }
        break;
    case 8:
        if (pressed) {
            forward_all(0.5);
        } else {
            stop_all();
        }
        break;
}
```

In case you haven't run into the **switch** statement before, it chooses a **case** block based on the value given to it, **buttnum** in this case. The case block with the
corresponding value is executed. Notice that each `case` block ends with a `break` statement. This exits the `switch`. If they were’t there, the next `case` block would be executed, and so on until the end of the `switch` was reached or a `break` statement was encountered.

The entire WobblyBot.ino is below.

```cpp
#include <stdarg.h>
#include <string.h>
#include <Arduino.h>
#include <SPI.h>
#include "Adafruit_BLE.h"
#include "Adafruit_BluefruitLE_SPI.h"
#include "Adafruit_BluefruitLE_UART.h"
#include "BluefruitConfig.h"
#include "Adafruit_Crickit.h"
#include "seesaw_servo.h"
#include "seesaw_motor.h"
#define FACTORYRESET_ENABLE         1
#define MINIMUM_FIRMWARE_VERSION    "0.6.6"
#define MODE_LED_BEHAVIOUR          "MODE"

uint8_t readPacket(Adafruit_BLE *ble, uint16_t timeout);
float parseFloat(uint8_t *buffer);
void printHex(const uint8_t *data, const uint32_t numBytes);

// the packet buffer
extern uint8_t packetbuffer[];

//#define DEBUG 1

Adafruit_BluefruitLE_SPI ble(BLUEFRUIT_SPI_CS, BLUEFRUIT_SPI_IRQ, BLUEFRUIT_SPI_RST);

// setup crickit

Adafruit_Crickit crickit;
seesaw_Servo legs[] = {seesaw_Servo(&crickit),
                      seesaw_Servo(&crickit),
                      seesaw_Servo(&crickit),
                      seesaw_Servo(&crickit)};
```

©Adafruit Industries
const int front_right = 0;
const int front_left = 1;
const int rear_right = 2;
const int rear_left = 3;

seesaw_Motor tail(&crickit);
float tail_power = 0.5;

// conditional output routines
void error(const __FlashStringHelper *err)
{
    digitalWrite(13, HIGH);
    #ifdef DEBUG
    Serial.println(err);
    #endif
}

void log(const __FlashStringHelper *msg)
{
    #ifdef DEBUG
    Serial.println(msg);
    #endif
}

// Motor Control
// Left and right motors turn in the opposite direction
const float motor_directions[4] = {+1.0, -1.0, +1.0, -1.0};

// pins to connect each servo to
const int servo_pins[4] = {CRICKIT_SERVO1, CRICKIT_SERVO2, CRICKIT_SERVO3, CRICKIT_SERVO4};

// PWM ranges for each motor, tune these so that setting the angle to 90 stops the motor
int pwm_ranges[4][2] = {{500, 2400}, {500, 2400}, {500, 2400}, {500, 2400}};

const __FlashStringHelper *leg_names[] = {F("Front right"), F("Front left"), F("Rear right"), F("Rear left")};

int speed_to_angle(float speed)
{
    return (int)(speed * 90.0 + 90.0);
}

void set_leg(int leg, float speed)
{
    int angle = speed_to_angle(speed * motor_directions[leg]);
    #ifdef DEBUG
    Serial.print(F("Setting "));
    Serial.print(leg_names[leg]);
    Serial.print(F(" to "));
    Serial.print(angle);
    #endif
    legs[leg].write(angle);
}

// Stop the listed motors
// -1 required as the last argument
void stop(int leg, ...)
{ va_list args;
  va_start(args, leg);
  log(F("Stop"));
  while (leg != -1) {
    set_leg(leg, 0.0);
    leg = va_arg(args, int);
  }
  va_end(args);
}

void stop_all()
{
  stop(front_right, front_left, rear_right, rear_left, -1);
}

void forward(float speed, ...)
{
  va_list args;
  va_start(args, speed);
  int leg = va_arg(args, int);
  log(F("Forward"));
  while (leg != -1) {
    set_leg(leg, speed * motor_directions[leg]);
    leg = va_arg(args, int);
  }
  va_end(args);
}

void forward_all(float speed)
{
  forward(speed, front_right, front_left, rear_right, rear_left, -1);
}

void reverse(float speed, ...)
{
  va_list args;
  va_start(args, speed);
  int leg = va_arg(args, int);
  log(F("Reverse"));
  while (leg != -1) {
    set_leg(leg, speed * -1 * motor_directions[leg]);
    leg = va_arg(args, int);
  }
  va_end(args);
}

void reverse_all(float speed)
{
  reverse(speed, front_right, front_left, rear_right, rear_left, -1);
}

void rotate_clockwise(float speed)
{
  forward(speed, front_left, rear_left, -1);
  reverse(speed, front_right, rear_right, -1);
}

void rotate_counterclockwise(float speed)
{ forward(speed, front_right, rear_right, -1); 
 reverse(speed, front_left, rear_left, -1);
}

void initialize()
{
 stop(front_right, front_left, rear_right, rear_left, -1);
}

void wag(float speed)
{
 #ifdef DEBUG
  Serial.print(F(“Wag ”));
  Serial.println(speed);
  #endif
  tail.throttle(speed);
  delay(75);
  tail.throttle(0.0);
  delay(50);
}

// Start things up
void setup()
{
  pinMode(13, OUTPUT);
  digitalWrite(13, LOW);
  #ifdef DEBUG
    while (!Serial); // required for Flora & Micro
    delay(500);
  #endif
  log(F(“WobblyBot”));
  log(F(“-----------------------------------------”));

  // Initialise the module
  log(F(“Initialising the Bluefruit LE module: ”));
  if (!ble.begin(VERBOSE_MODE))
  {
    error(F(“Couldn’t find Bluefruit, make sure it’s in CoMmanD mode & check wiring?”));
  }
  log( F(“OK!”) );

  if ( FACTORYRESET_ENABLE )
  {
    // Perform a factory reset to make sure everything is in a known state
    log(F(“Performing a factory reset: ”));
    if (! ble.factoryReset() ){
      error(F(“Couldn’t factory reset”));
    }
  }

  // Disable command echo from Bluefruit
  ble.echo(false);

  log(F(“Requesting Bluefruit info:”));
  // Print Bluefruit information
  ble.info();
log(F("Please use Adafruit Bluefruit LE app to connect in Controller mode"));
log(F("Then activate/use the sensors, color picker, game controller, etc!
"));

ble.verbose(false); // debug info is a little annoying after this point!

// Wait for connection
while (!ble.isConnected()) {
  delay(500);
}

log(F("**********************"));

// LED Activity command is only supported from 0.6.6
if (ble.isVersionAtLeast(MINIMUM_FIRMWARE_VERSION)) {
  // Change Mode LED Activity
  log(F("Change LED activity to " MODE_LED_BEHAVIOUR));
  ble.sendCommandCheckOK("AT+HWModeLED=" MODE_LED_BEHAVIOUR);
}

// Set Bluefruit to DATA mode
log(F("Switching to DATA mode!") );
ble.setMode(BLUEFRUIT_MODE_DATA);

log(F("**********************"));

if (!crickit.begin()) {
  error(F("Error initializing CRICKIT!") );
}

log(F("Crickit started"));

for (int leg = 0; leg < 4; leg++) {
  legs[leg].attach(servo_pins[leg], pwm_ranges[leg][0], pwm_ranges[leg][1]);
}

tail.attach(CRICKIT_MOTOR_A1, CRICKIT_MOTOR_A2);

// Fill these functions in with the movement scripts you want attached to
// the controller's 1-4 buttons

void demo1()
{
  forward_all(0.5);
  delay(5000);
  rotate_clockwise(0.5);
  delay(2000);
  forward_all(0.75);
  delay(4000);
  rotate_counterclockwise(0.5);
  delay(3000);
  stop_all();
}

void demo2()
{
}

void demo3()
{
}

void demo4()
{
```c
void loop()
{
  wag(tail_power);
  tail_power *= -1.0;

  // Wait for new data to arrive
  uint8_t len = readPacket(&ble, BLE_READPACKET_TIMEOUT);
  if (len == 0) return;

  // Got a packet!
  // printHex(packetbuffer, len);

  // Buttons
  if (packetbuffer[1] == 'B') {
    uint8_t buttnum = packetbuffer[2] - '0';
    boolean pressed = packetbuffer[3] - '0';

    #ifdef DEBUG
    Serial.print("Button "); Serial.print(buttnum);
    if (pressed) {
      Serial.println(" pressed");
    } else {
      Serial.println(" released");
    }
    #endif

    switch(buttnum) {
      case 1:
        if (pressed) {
          demo1();
        }
        break;
      case 2:
        if (pressed) {
          demo2();
        }
        break;
      case 3:
        if (pressed) {
          demo3();
        }
        break;
      case 4:
        if (pressed) {
          demo4();
        }
        break;
      case 5:
        if (pressed) {
          rotate_counterclockwise(0.5);
        } else {
          stop_all();
        }
        break;
      case 6:
        if (pressed) {
          rotate_clockwise(0.5);
        } else {
          stop_all();
        }
        break;
      case 7:
        if (pressed) {
          reverse_all(0.5);
        }
    }
  }
}
```
} else {
  stop_all();
} break;

case 8:
  if (pressed) {
    forward_all(0.5);
  } else {
    stop_all();
  }
  break;
}
}

---

**Micro:bit and MakeCode**

For a final version we'll use a BBC micro:bit and the CRICKIT for micro:bit. We'll also write the code using MakeCode.

If you are new to Microsoft MakeCode, you can [learn the basics of MakeCode here](https://adafruit.it/wWd).

To add CRICKIT support, MakeCode needs to load an extension - to bring in some blocks that are not usually loaded in the main program. A detailed walk-through of how to do this for the micro:bit as well as a discussion of the CRICKIT blocks that the extension provides is [here in the CRICKIT guide](https://adafruit.it/BKC) - scroll down to the "For micro:bit Crickit" section. You will need to use the beta version of MakeCode for micro:bit and add the extension web address manually. But it works well.

If you've read through the Python and/or C++ pages, the MakeCode program should look similar, but much simplified. For example, here is the `forward` function:
The stop function is a bit different. Instead of setting each servo to 0%, it sets the pwm pulse width. Remember that servos are all a bit different and can respond differently to an exact pulse width. In the Python and C++ libraries we can set the pulse ranges and stop will be in the middle. So changing the range will move the stop point. We don't have that capability in the CRICKIT MakeCode extension, so to stop we can set the pulsewidth directly. With a little trail and error we can find the setting for each servo that causes it to stop.
Two `on button - pressed` blocks are used to trigger one of two predefined movement scripts. Here's an example:
Here's the entire program, as well as a link to open it in MakeCode.
Wrapup

In this guide we built a little robot with a simple locomotion system that gives it some quirky emergent behavior due to the inherent asymmetry of continuous servos. It can provide an interesting base for continued experimentation.

Beyond that, this project shows one feature of the CRICKIT now that there are 3 models to choose from (and a Raspberry Pi version on the way): they’re interchangeable. You can start a project with a micro:bit. If that becomes too constraining, or you want to try using CircuitPython, you can switch to the Circuit Playground Express version. If you want to get your project online, you can switch to the Feather version and use a Feather M0 WIFI (https://adafru.it/n6f) or a Feather Huzzah ESP8266 (https://adafru.it/n6A).
If we were to switch to using the FeatherWing CRICKIT and a Feather M4 Express (https://adafru.it/Cmy), we would have plenty of computational power to explore some AI techniques to add autonomous control. The build leaves a lot of the CRICKIT’s capabilities unused. We've used none of the digital/analog I/O lines, or the speaker output, or NeoPixel output, or the cap-touch inputs. There’s also a motor output not being used, and the 4 drive outputs. Thanks to the use of the upper deck to mount the controller+CRICKIT, there's room on the chassis to add more stuff: lights, a speaker, sensors, etc.

If you take this project and do something with it, please share with the community on the Adafruit Discord (https://adafru.it/BYI) or the weekly Show and Tell (https://adafru.it/dfD)!