



BOTTANGO ARDUINO DRIVER DOCUMENTATION

Documentation rev 11

End User License Agreement	4
ALL USE OF BOTTANGO IS AT YOUR SOLE RISK.	4
Hardware and Driver Beginner Documentation	5
Section 1 - Beginner Hardware Details	5
I'm totally new to building robots, and need somewhere to start!	5
My Servo /Stepper /Motor isn't moving and I'm new to all this!	6
Set Up a Microcontroller for Bottango	7
A Little Background	7
You may be able to upload the driver using the desktop app	7
What do you need to upload via the Arduino IDE	8
Uploading the Bottango software on to your Arduino	8
You're Good to Go!	10
Uh oh! Compilation Error	10
Don't Forget To Update Your Microcontroller!	10
What Boards are Compatible with Bottango?	10
What Boards Do You Recommend?	11
How To Safely Power Servos	12
A Microcontroller Is Not a Motor Power Circuit	12
Build Your Own Simple Starting Servo Power Circuit (1-2 small servos)	12
More than 2 small servos	13
Adafruit PCA9685 16 Channel Servo Driver - Pros and Cons	14
Adafruit PCA9685 16 Channel Servo Driver - Using It in Bottango	15
Controlling Lots of Motors	17
The Limits of a Single Arduino Uno R3 - 8 Motors Max!	17
Split Your Motors Across Multiple Microcontrollers	17
Switch to a more powerful microcontroller - 16 out of the box	18
Need more than 16 on one microcontroller?	18
Using an ESP32 with Bottango	19
What ESP32 dev boards can I use?	19
Learn your Pinouts	20
Setup an ESP32 to use with Bottango	20
Advanced Documentation and Features	21

The rest of this documentation	21
The Modules File & Playing Exported Animations	22
How To Enable/Disable Modules	22
Playing Exported Animations	23
Playing Exported Animations - Save To Code	23
Exported Code Animations - Memory Limitations	23
Playing Exported Animations - Save To SD Card	24
Adding your own playback logic	24
Custom Events, Motors, and Other Callbacks	25
What is the Callbacks File?	25
Driver Lifecycle Events	25
Custom exported animation playback logic	25
Controlling the desktop app from your driver	26
What are Custom Motors and Events	26
Effector Identifier	27
Effector Lifecycle Events	27
Auto Syncing Stepper Motors	29
Putting the Lifecycle Events Together for Custom Motors	30
Responding to Custom Events	30
Using the Bottango Networked Driver	34
What is the Bottango Networked Driver?	34
When is the Bottango Networked Driver the right choice for me?	34
Starting a Bottango Network Server	35
Running a Network Client	35
Modifying the example Python Code	35

-1-

End User License Agreement

ALL USE OF BOTTANGO IS AT YOUR SOLE RISK.

BOTTANGO IS IN BETA TESTING AND MAY CONTAIN ERRORS, DESIGN FLAWS, BUGS, OR DEFECTS. BOTTANGO SHOULD NOT BE USED, ALONE OR IN PART, IN CONNECTION WITH ANY HAZARDOUS ENVIRONMENTS, SYSTEMS, OR APPLICATIONS; ANY SAFETY RESPONSE SYSTEMS; ANY SAFETY-CRITICAL HARDWARE OR APPLICATIONS; OR ANY APPLICATIONS WHERE THE FAILURE OR MALFUNCTION OF THE BETA SOFTWARE MAY REASONABLY AND FORESEEABLY LEAD TO PERSONAL INJURY, PHYSICAL DAMAGE, OR PROPERTY DAMAGE.

YOU MUST REVIEW AND AGREE TO THE BOTTANGO BETA SOFTWARE END USER LICENSE AGREEMENT BEFORE USING BOTTANGO: <http://www.Bottango.com/EULA>

-2-

Hardware and Driver Beginner Documentation

Section 1 - Beginner Hardware Details

The large Bottango Documentation pdf that comes in the main folder of the downloaded Bottango application covers in great detail how to use the Desktop app. But in order to control a robot in real life with Bottango, you need not just the desktop application, but some properly working real world hardware as well. This documentation helps to get your hardware up and running.

In this first section, we'll cover a lot of beginner topics. We'll explain how to set up a microcontroller with the provided Bottango Arduino Driver. We'll also address some common questions around robot building and hardware that have become frequent pain points of new robot builders.

If you are just starting out, I recommend you read this first section, and then stop there. The second section of this documentation will be advanced topics for those that have experience building robots and/or want to explore the code of the driver itself and other advanced topics.

I'm totally new to building robots, and need somewhere to start!

Bottango helps you control a robot, but it doesn't help you BUILD a robot! So if you're coming entirely from square one, it's best to take a step back, and learn a bit of the fundamentals of robot making.

One of the best places to start is with an Arduino starter kit. The official one from the Arduino company contains all you'd need to get a moving servo (and more) and also has some really great tutorials. There are more affordable options, but in my opinion though the parts themselves are around the same quantity and quality, the quality of the included documentation and tutorials is lower. You can decide which option is right for you.

Your goal is to learn how to move a servo with an Arduino WITHOUT Bottango. If you want to build your own "starter kit," you would just need an Arduino Uno, a servo, a breadboard, a 5v power supply, and some jumper wires. There's lots of youtube videos and tutorials online for getting a servo moving with an Arduino. Once you understand enough to get a single servo moving, then you can add Bottango in, and animate that servo with Bottango. You can use Bottango to craft a custom animation for your servo, synchronize it with sound, etc.

The best place to start learning Bottango is in the documentation that comes with the downloaded app. "BottangoDocumentation.pdf" That documentation covers everything about how the app works. However, there's also an early chapter called "A Crash Course in Bottango" that runs through all the big ideas of how to use the app in one short chapter. I'd start there if you want to hit the ground running.

A good first project is to make a mouth open and close using a single servo, and to animate it along with some audio of speech. Once you've gotten that far, you're off to the races, and can keep building on what you've learned. Good luck!

My Servo / Stepper / Motor isn't moving and I'm new to all this!

Bottango is a tool to control a robot, but it is not designed to troubleshoot or build the hardware itself.

If your motor does not seem to be working with Bottango, the first thing to find out is if the motor and your setup works at all. The most effective way to get things working is to take a step back, and remove Bottango from the equation for a bit. You'll want to get your hardware (motor, Arduino, etc.) working WITHOUT Bottango, and then add Bottango back in when you're confident in your set up and ready for more advanced motion control.

One recommendation is to look for Arduino tutorials that use the kind of motor you have (Servo, stepper, etc.), wire everything up as explained in these tutorials, and control it using the tutorial provided code uploaded through the Arduino IDE. Once you are certain that your electronics work as expected, then controlling them with Bottango should be an easier evolution. Some Google / YouTube searching will get you a long way!

-3-

Set Up a Microcontroller for Bottango

A Little Background

Bottango takes two parts to work: the included Bottango application, and at least one microcontroller for the Bottango application to communicate with. The Bottango application sends commands to a microcontroller over a serial connection, and the microcontroller then moves motors, etc. as required.

In order to provide what you need for both parts, we supply this Arduino-compatible code in addition to the Bottango application. For all out-of-the-box functionality of Bottango, you shouldn't need to edit or modify the included Arduino-compatible code.

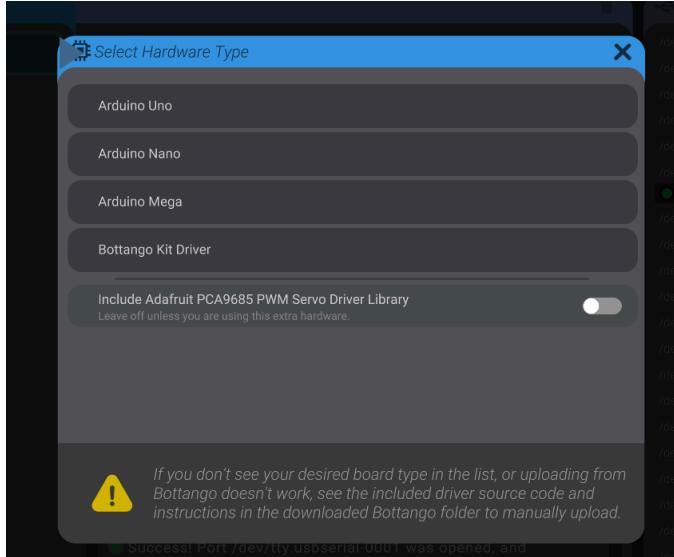
That being said, the Arduino-compatible code is provided to you open source (see `BottangoArduinoLicence.txt`). If your needs require you to edit the included code, you are able to do so.

Your first step will be to get the Bottango Arduino Driver on to your Microcontroller.

You may be able to upload the driver using the desktop app

If you are using an Arduino Uno R3, Nano, or Mega, you may not need to use this guide or the driver source code. You can use the desktop app to upload a stock version of the driver to those kinds of boards. After familiarizing yourself somewhat with the Bottango Application, and beginning to read through the desktop app documentation (read chapter 7 at a minimum for these steps), you can follow these steps:

- 1** Connect your microcontroller over USB, open Bottango, and click "New Studio Project"
- 2** Go to "Hardware" tab, and then the "Drivers" subtab
- 3** Connect to the port your microcontroller is on.
- 4** Click the "Upload Bottango Driver" button once connected to that port.



You'll be shown a menu with the option to enable support for the PCA9685 driver if required, and then click the kind of board you're trying to upload to. It will handle uploading from there.

If you want to manually upload the Bottango driver yourself from the source code, use a board not supported by the desktop app for uploading, or make changes to the driver, read on from here.

What do you need to upload via the Arduino IDE

In order to set up your Arduino with Bottango using the driver source code, you will need the following:

- An Arduino compatible microcontroller, and a USB cable.
- The Arduino IDE installed on your computer (<https://www.arduino.cc/en/software>).
- The BottangoArduino.ino Arduino sketch and associated files, included in the same folder as this documentation.

Uploading the Bottango software on to your Arduino

- 1 Open the BottangoArduino.ino file, which should open in the Arduino IDE if you have it installed.

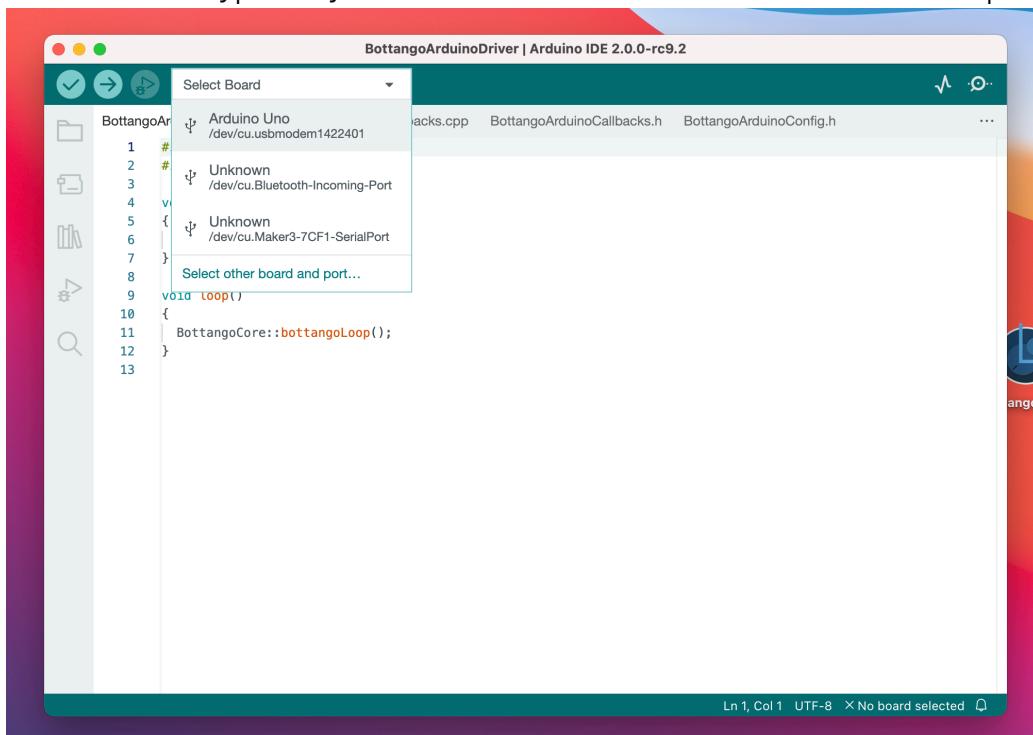
You will see the BottangoArduino.ino file opened:

```

1 #include "src/BottangoCore.h"
2 #include "src/BasicCommands.h"
3
4 void setup()
5 {
6     BottangoCore::bottangoSetup();
7 }
8
9 void loop()
10 {
11     BottangoCore::bottangoLoop();
12 }
13

```

- 2** Connect your Arduino compatible microcontroller to your computer via USB.
- 3** Select the board type for your microcontroller in the “Select Board” dropdown.



If you’re not sure what board you have, you probably have an Arduino Uno. If your board looks larger than an Arduino Uno, it’s probably an Arduino Mega, and if it looks smaller than an Arduino Uno, it’s probably

an Arduino Nano. If it's not showing up in the list when you connect it to your computer via USB, you may have an unusual or off brand board, and will need to troubleshoot detecting it by the Arduino IDE. Feel free to join our Bottango Discord for advice from the community.

- 4** Click the “upload” right pointing arrow icon in the Arduino IDE to upload the Bottango Arduino code to your microcontroller.



You're Good to Go!

If everything worked right, you should have the microcontroller set up to work with Bottango. Continue to refer to the main documentation on how to use Bottango to control your robots, and this section in this documentation for more hardware knowhow.

If you’re having trouble, here’s some helpful documentation from Arduino: <https://www.arduino.cc/en/Guide/Windows> or <https://www.arduino.cc/en/Guide/MacOSX>. As well, you can join the Bottango Discord group for support: <https://discord.gg/6CVfGa6>

Uh oh! Compilation Error

If the unchanged provided source code is giving you a compilation error (something like “srcBottangoCore.h No such file or directory”) the most common reason is you are trying to upload the driver from inside a .zip file on Windows. On Windows, instead of double clicking the .zip file for the Bottango installer and application, right click the .zip file, extract the contents, and work within the extracted folder. The Bottango Arduino Driver source code does not work when it’s inside the compressed zip, it needs to be extracted first.

Don't Forget To Update Your Microcontroller!

When you download an update to Bottango, you should always repeat the above steps to update the code on your microcontroller as well! Bottango is in heavy development so when the application changes, a lot of times the microcontroller code changes with

What Boards are Compatible with Bottango?

Bottango comes with an open source C++, Arduino compatible driver to run on your hardware. The very short answer is that most Arduino compatible microcontrollers that have a servo library are likely compatible and can be used with Bottango.

What Boards Do You Recommend?

If you're just starting out, an Arduino Uno R3 is plentiful, low cost and has years and years of community support. It is perhaps the most popular single development board model used with Bottango, and you'd be perfectly reasonable to start with one of those. In fact, if you've been at all interested in hobby electronics, you likely already have one. There are limitations with using an Arduino Uno R3 and Bottango though, see the chapter on "How do I control lots of servos."

If you want to use something more capable than an Arduino Uno R3, my go to recommendation is an ESP32 development board. They are also plentiful, around 100x more capable than an Uno R3, and often cheaper too! The downside is that they are less standardized so there's a bit more learning at the start. See the chapter on ESP32 and Bottango for more details.

Otherwise, some other options that users have chosen are ESP8266, Teensy, Arduino Zero, Arduino Nano 33 Family, etc. Much more development boards work with Bottango than don't.

Keep an eye out on Bottango's website too! We're developing our own general purpose animatronic control boards that will be a great option once available.

-4- How To Safely Power Servos

A Microcontroller Is Not a Motor Power Circuit

Safely powering servos is one of the earliest and biggest challenges you'll encounter in your journey on robot building if you're just starting out. It's not insurmountable!

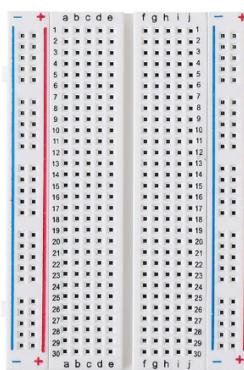
When you're just starting out, it's going to be very tempting to power your servos directly off the microcontroller (Arduino Uno, etc.) itself, connected to your computer. Unfortunately, that approach is doomed to failure. As soon as a servo draws more than even a small amount of current, your laptop is going to shut down that usb port at best, or get damaged at worse. As an example, an Arduino Uno R3 is rated at 150mA on the VIN pin, and even a single small servo can draw 1A (1000ma), and a large servo can easily draw 3A (3000ma) at stall.

When your servo draws too much power, your microcontroller will shut down as well since it's sharing power with the servo. A microcontroller development board like an Arduino Uno R3 is just not meant to or designed to power servos, it's really only meant to create the signal that controls those servos. And first and foremost, you want to do it SAFELY!

Build Your Own Simple Starting Servo Power Circuit (1-2 small servos)

If you're just starting out and want to learn the basics of powering servos, you can build your own power circuit with some low cost components.

One of the easiest ways to get started is with a breadboard. A breadboard can handle 1-2a current, and power 1-2 small servos. Google will be your friend here as well as you learn the basics of circuit building with a breadboard.



You'll need to get power onto the breadboard.

One option is a AA battery holder, that has 4 AA's (6v total).

Another Option would be a small 5v2a DC power supply and a DC screw terminal adapter. Take very careful note of the voltage of a power supply if purchasing one for servo power. You'll want to make sure that the voltage of your power supply is within the required voltage input of a servo. Most servos happily take 5v. A select few special high voltage servos can take higher voltage than 6v, never input power above 6v into a servo unless you are confident it is a servo that is designed for higher voltage.

Here's an example AA battery holder: <https://www.adafruit.com/product/3859>

Here's an example DC screw terminal adapter you would use with a 5v2a power supply: <https://www.adafruit.com/product/368>

You would connect the power to the breadboard, and then the ground and signal pins from your microcontroller to the breadboard. Then you would take power and signal on the servo from the breadboard after making those connections.

This is a good approach at the very beginning, but it doesn't scale. You can't safely power more than 1 or 2 small servos this way, but it's a good way to learn the basics.

More than 2 small servos

If you want to stick with building your own power circuit, you'll need to build your own board that can handle more than 1-2A current. Learning to solder, and making something out of proto/perf board and larger wires will be your path forward.

There are also easy products you can buy that make learning and building your own solution not needed.

To start, Bottango sells a 12 channel servo shield that is rated up to 7A of power:
<https://www.bottango.com/products/bottango-servo-shield>



The Bottango Servo shield plugs directly into an Arduino Uno R3 or Arduino Mega, and has a separate power jack for powering servos separately and safely from the Microcontroller. You'll need to bring your

own power supply which feeds the power directly to the servos, so make sure it's a voltage compatible with your servos (5v is usually a safe choice). See the product web site for more details on this option.

Servo City has a Servo power distribution board as well:

<https://www.servocity.com/8-channel-servo-power-node/>

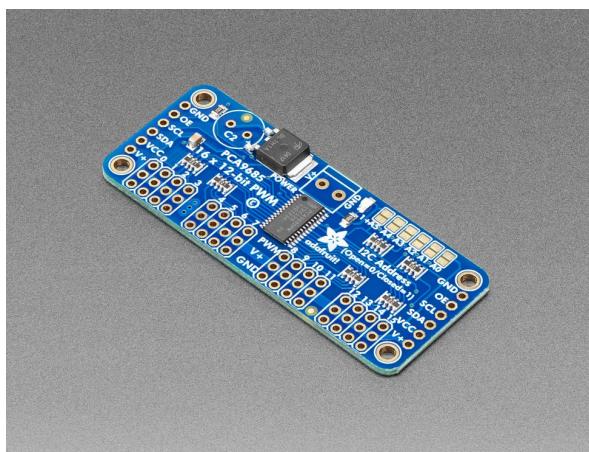


There are international distributors for their products, so you can find them around the world. Note that the power input for this board is an XT30 connector, so you'll need a power supply that can output to that port. You can purchase a compatible power supply from Servo City or get a converting adapter.

Adafruit PCA9685 16 Channel Servo Driver - Pros and Cons

A popular option in the hobby servo world is the Adafruit PCA9685 Servo Driver:

<https://www.adafruit.com/product/815>



It has a servo power circuit built into it, as well as extra hardware to generate the servo signals external to your microcontroller, using a communication protocol called i2c.

Because the hardware design is open source for this board, there are a lot of low cost copies made by other manufacturers. A common flaw in some low cost copies is they choose a very low current component in part of the power circuits (the reverse polarity mosfet) **that overheats and burns out around 4A of current**. For safety reasons, and to ensure working hardware, I recommend sticking with the official board manufactured by Adafruit themselves for this reason, or ensuring that you do not select a low cost copy that has that flaw.

Though the PCA9685 16 channel servo driver is supported in the Bottango software, there are some pros and cons to using it with Bottango. In general, if you can get away with it, having a standalone power circuit, and using the microcontroller itself rather than using a PCA9685 to generate the signals for the servos is going to be easier and better performance with Bottango.

Due to additional resource usage, on an Arduino Uno R3 you can only control up to 6 max servos with a PCA9685, less than the 8 max without it. A more powerful microcontroller can use all 16 channels without effecting performance, like an ESP32. A mega can handle all 16 channels, but it will have added latency using the i2c communication to the board vs directly using the pins on the mega and not using a PCA9685.

Finally, you'll need to wire up the PCA9685's i2c connections correctly to use it, which can be intimidating to folks just starting out. Follow the guide from Adafruit on how to wire up: <https://learn.adafruit.com/16-channel-pwm-servo-driver>. There are a lot of incorrect guides floating around on the internet for wiring that will cause issues / damage hardware. The official guide linked is the correct way.

For this reason, the common power flaw in low cost copies, etc. I generally don't recommend it as a good option for beginners.

PROS:

- Servo Power Circuit Built In
- Supported in Bottango
- Theoretically up to 16 servos
- Popular and plentiful

CONS:

- Can be hard to wire up i2c pins if you're new
- Low cost options often have nasty power circuit flaw
- Takes resources and adds latency on more resource strapped microcontrollers (Uno R3, Mega) etc.
- Can control less servos than not using it on an Uno R3
- Some online tutorials recommend incorrect wiring

Adafruit PCA9685 16 Channel Servo Driver - Using It in Bottango

If you decide want to use the Adafruit PCA9685 16 Channel Servo Driver in Bottango, you'll need to take a few extra steps.

1) You'll need to enable support for the Adafruit PCA9685 16 Channel Servo Driver in the Bottango Driver code, it is disabled by default. If you are using an Arduino Uno R3, Nano, or Mega, you can reupload the driver to your board in the Desktop application, and enable support for the PCA9685 board directly in the app. For details, see Chapter 7 in the Bottango Documentation. The section titled "Driver status Details" has step by step instructions for uploading the driver in the desktop app.

If you want to enable it in the driver source code and reupload in the Arduino IDE, you'll need to enable it in the modules file, as well as install the required library in the Arduino IDE library manager. See the chapter in this documentation on the driver modules file for how to enable that module in the driver.

2) In the desktop app, when selecting the pin for your servo, change the connection type dropdown from "pin" to "i2c and pin." The i2c address will be the address of the pca9685 driver. Unless you have soldered a new address on to the board, it will be what the default is in Bottango (you'd know if you've done this). The pin will be the channel on the board your servo is connected to.

-5-

Controlling Lots of Motors

The Limits of a Single Arduino Uno R3 - 8 Motors Max!

If you're using an Arduino Uno R3 with Bottango, there are some limitations you need to be aware of. (Note, an Arduino Nano is effectively identical to an Uno R3 in terms of Bottango usage. Everything said about an Uno R3 applies to a Nano throughout this documentation)

Let's compare an Arduino Uno R3 to a modern computer that has, for an example, 16 gigabytes of ram and an 8 core processor running at 2.5 GHz. An Arduino Uno R3 has 2kb ram and a single core running at 16 MHz.

If it's not immediately apparent the **huge** difference in power, the modern computer has 16 MILLION kb ram compared to the Arduino's two. And the modern computer's processor is as much as 125,000% faster than the Arduino's.

All of this is to set your expectations of just how much a single Arduino Uno R3 can do. Bottango's code is well optimized, but it is fairly intensive in amount of processing for an average Arduino program. As such, in this chapter we'll go over the best practices of getting great performance, and how to get the results you want.

In short, a single Arduino Uno can only control a maximum of 8 motors with Bottango.

Split Your Motors Across Multiple Microcontrollers

If you want to stick with an Arduino Uno R3 and you want to control lots of motors, one of the easiest ways to build what you need is to just split the motors across multiple microcontrollers.

Bottango can control a lot of Arduinos, but a single Arduino can only control a fixed number of motors. Bottango is designed to allow you to easily split your motors across multiple Arduinos.

Bottango has no problem communicating with multiple Arduinos at once, and in fact is optimized to do just that! If your build can facilitate it, and you want to stick with an Arduino Uno R3, just get one per 8 motors needed.

This chart is the recommended number of motors to put on a single Arduino Uno R3:

Motor Type	Ideal	Allowed	Requires Modifications
Servo	6 or Less	8 or Less	9 or More

Motor Type	Ideal	Allowed	Requires Modifications
Custom Motor / Event	6 or Less	8 or Less	9 or More
Stepper	3 or Less	8 or Less	9 or More

What happens if you get out of ideal and into the “Allowed” zone on an Uno R3? You may see more sluggishness or choppiness, depending on the complexity of the animations being sent to the Arduino (in terms of number of unique keyframes). Depending on your needs you might not see any difference, or it might be small enough to not matter.

Stepper motors especially will get more choppy the more you add to the Arduino. Even a single stepper motor on an Uno R3 does not run very smoothly with Bottango. If you’re serious about using stepper motors, I strongly recommend using an ESP32 or a similarly powerful microcontroller.

Switch to a more powerful microcontroller - 16 out of the box

There’s a lot to love about the classic Arduino Uno R3. It has years and years of support and community built around it. But as the years have gone by, there are a lot of newer options with way more capability, often at costs equal to or even lower than an Arduino Uno R3.

For the most basic “level up” from an Arduino Uno R3, you can move to an Arduino Mega. With an Arduino Mega (and all microcontrollers that are not an Arduino Uno R3) you’ll be able to control up to 16 motors on your microcontroller. The driver will detect automatically for you and raise the limit to 16 for you. A mega will start to get a little laggy as you get closer to 16, but it will for the most part keep up.

My go to recommendation for Bottango though is an ESP32 development board. With an ESP32 you can control 16 (or even more!) motors without breaking a sweat. An ESP32 has maybe 100x the performance capabilities of an Arduino Uno R3 and is often a lower price. There’s a few extra steps you’ll need to take to get it up and running (see the chapter on ESP32), but it’s easy and worth it in my opinion!

Need more than 16 on one microcontroller?

If you want to go past the maximum 16 motors per microcontroller and are confident your microcontroller can keep up, you certainly can, but you’ll need to make a few changes. In the configuration file of the driver, update the max effectors field (see the chapter on the config and modules file) to the number of your choice. But proceed only if you understand what you’re doing. There’s not a lot of use cases (but also not zero) where more than 16 on a single microcontroller makes sense versus splitting the work amongst multiple boards.

-6-

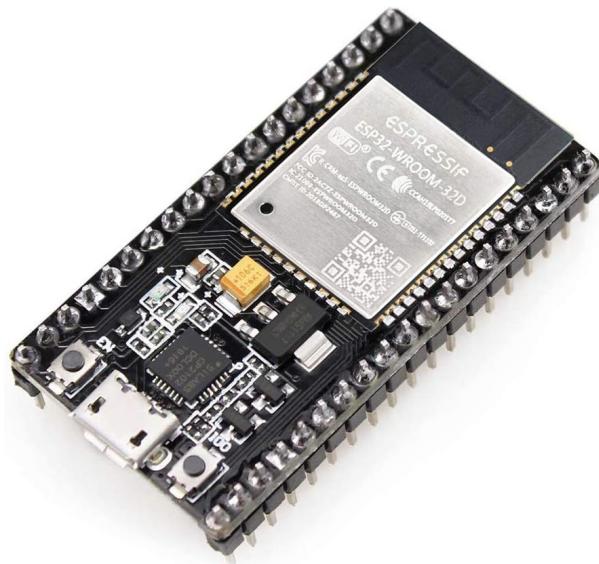
Using an *ESP32* with Bottango

What *ESP32* dev boards can I use?

ESP32 is a popular, powerful, and affordable microcontroller that has become quite popular. Besides having great specs for the cost, it also has great features like bluetooth and wifi. Outside of an Arduino Uno R3, an ESP32 based dev board is probably the second most popular choice of microcontroller to use with Bottango.

Why use an ESP32 rather than an Arduino Uno R3? Pretty much everything an Arduino Uno R3 can do, an ESP32 can do 100 times faster. An esp32 has 520k RAM to the Uno's 2k, 4MB storage for animations to the Uno's 32K, and 240mhz to the Uno's 16mhz. What's more, an ESP32 is often lower cost than the Uno R3.

However, unlike an Arduino Uno R3, there is not as much standardization across different dev boards, and there are different underlying chipsets. For the most compatible and easiest to use choice, select an ESP32 based dev board that uses an ESP32-WROOM-32E or ESP32-WROOM-32D chip. When you look at the dev board, you'll see the chip model printed on the microcontroller:



This example has an ESP32-WROOM-32D. (D vs E is just the revision. E is newer, but not different in a way that will likely matter to use with Bottango). If you're new to ESP32, you'll want to avoid ones that are named something like ESP32-S3 or ESP32-C6. Those are newer chips that have additional functionality

you likely won't use but also more compatibility issues with Bottango and Arduino code. It is possible to use them, but I would only recommend it to advanced users.

Learn your Pinouts

Every Arduino Uno R3 has the same layout and pinouts. Pin 12 is in the same spot in all Uno R3s. However, there is not that level of standardization on an ESP32. You'll need to make good use of the pinout diagram that comes with the board you purchase, as well as double check it against the pin names printed on the board. I have also encountered some ESP32 where the pinouts are not as shown in the diagram/printed on the board correctly, so it's something to keep an eye out on. A multimeter and a blink sketch can help you identify your pins if you suspect they're not as printed.

Setup an ESP32 to use with Bottango

Unlike an Arduino Uno/Nano/Mega, you can't upload the driver to an ESP32 using the desktop app and you'll need to use the Arduino IDE to get it set up.

- 1 You need to set up the Arduino IDE to be able to upload code to an ESP32. Here's a great tutorial: <https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/>

- 2 You only need to do this one time.
After that's done, make sure you can upload Arduino sketches to your connected ESP32. Use a basic blink sketch, and if your ESP32 dev board has some kind of user LED (most do) set the pin on the blink sketch to your ESP32's user LED. Some ESP32 dev boards need you to hold down the "BOOT" button, then press the reset button before uploading new code onto it, but not all.

- 3 The Bottango Arduino Driver has checks to automatically configure itself for an ESP32. You will however need to install one extra library. In the Arduino IDE go to the library manager (Tools > Manage Libraries...) and search for "ESP32Servo." Install the result named ESP32Servo with Kevin Harrington as the author.

- 4 Select the port and board type. If you are using the kind of ESP32 recommended above (ESP32-WROOM-32E, etc.) the board type will be "esp32 > Esp32 Dev Module."

You can then upload the Bottango Arduino driver to your ESP32 the same as any other board type.

-7-

Advanced Documentation and Features

The rest of this documentation

From here we will cover some advanced topics. You'll learn how to modify the Bottango driver, including enabling and disabling extra functionality. We'll talk about how to extend the functionality of the driver, and the workflow for playing exported animations from the Bottango application.

If you're interested in adding more functionality outside of the built in features and a live USB connection to the Bottango desktop app, read on!

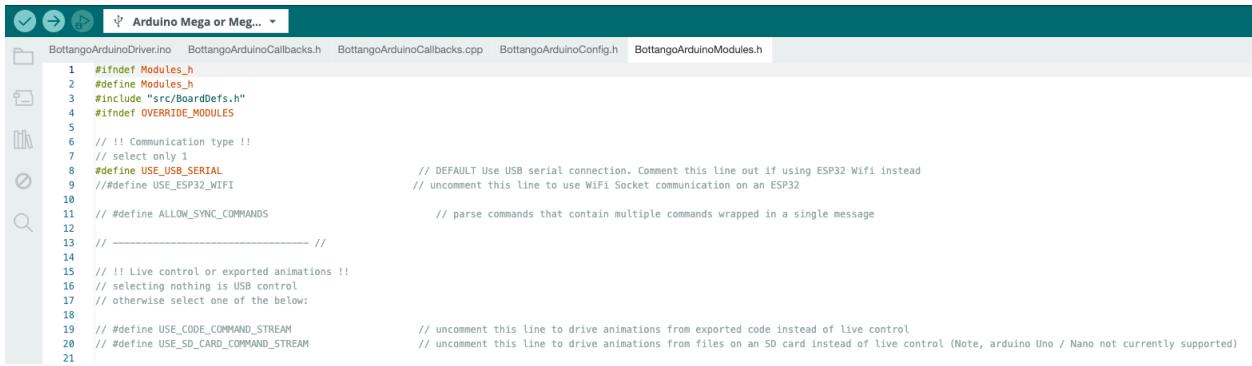
-8-

The Modules File & Playing Exported Animations

How To Enable/ Disable Modules

Once you open BottangoArduinoDriver.ino in the Arduino IDE, you should see multiple tabs. One of those tabs is named "BottangoArduinoModules.h"

You can use this file to turn on and off special functionality. Once you have made the changes you want, you'll need to recompile and reupload the driver from he Arduino IDE with these changes made. The desktop app can only upload the stock, unchanged version of the driver with the default modules.



```

Arduino Mega or Meg... ▾
BottangoArduinoDriver.ino BottangoArduinoCallbacks.h BottangoArduinoCallbacks.cpp BottangoArduinoConfig.h BottangoArduinoModules.h

1 #ifndef Modules_h
2 #define Modules_h
3 #include "src/BoardDefs.h"
4 #ifndef OVERRIDE_MODULES
5
6 // !! Communication type !!
7 // select only 1
8 #define USE_USB_SERIAL
9 // #define USE_ESP32_WIFI
10
11 // #define ALLOW_SYNC_COMMANDS
12 // parse commands that contain multiple commands wrapped in a single message
13 // -----
14
15 // !! Live control or exported animations !!
16 // selecting nothing is USB control
17 // otherwise select one of the below:
18
19 // #define USE_CODE_COMMAND_STREAM
20 // #define USE_SD_CARD_COMMAND_STREAM
21
// DEFAULT Use USB serial connection. Comment this line out if using ESP32 Wifi instead
// uncomment this line to use WiFi Socket communication on an ESP32
// uncomment this line to drive animations from exported code instead of live control
// uncomment this line to drive animations from files on an SD card instead of live control (Note, arduino Uno / Nano not currently supported)

```

Each line that starts with a "#" represents a module. If there is a "://" before the "#" symbol, that means that the module is turned off.

In the default driver, you'll notice that only two modules are enabled:

```
#define USE_USB_SERIAL
#define AUDIO_TRIGGER_EVENT
```

If you want to enable another module, you need to remove the "://" before the "#" symbol.

As an example, to enable support for the PCA 9685 Adafruit servo driver, you would delete the "://" in the line:

```
// #define USE_ADAFRUIT_PWM_LIBRARY
```

To look like:

```
#define USE_ADAFRUIT_PWM_LIBRARY
```

Once you save and re-upload, the driver will include that extra functionality. (Note that in the case of this module, you'll also need to install the required library "Adafruit PWM Servo Driver Library" in the Arduino IDE library manager).

Playing Exported Animations

There are a lot of modules in the modules file, but by far the most common you will use are the modules that switch the driver to reading exported animations instead of streaming animations over USB from the desktop app.

If you want to learn more about the other modules not covered in this documentation, ask on the Bottango Discord server. They are mostly used for internal development and Bottango hardware, but you are welcome to learn and use them as well.

Playing Exported Animations - Save To Code

When you export animations for the desktop app in "Save To Code" format, you will get two files per driver: "GeneratedCodeAnimations.cpp" and "GeneratedCodeAnimations.h"

Drag both of those files into the root of the BottangoArduinoDriver folder. They contain all the data needed to play the animations, you will not need to make any changes to them. As well, they contain the playback configuration like which animation to play on startup if any, etc.

In the Modules file, change:

```
// #define USE_CODE_COMMAND_STREAM
To
#define USE_CODE_COMMAND_STREAM
```

Which will enable the module that looks for animation data in the files you added rather than over USB. Re-upload and your driver will now ignore USB connections and use the exported code animations.

Exported Code Animations - Memory Limitations

When animations are exported, the size of the data is based on the number of keyframes, not the length of the animation. One animation with 10 total keyframes will take the same amount of space regardless of how long the animation is.

However, be aware of the limitations of how much space is available on your microcontroller. An Arduino Uno R3 usually has only around 10k of space left for animation storage after the driver itself is included. That could be only a minute or less of animation data, depending on how many keyframes.

An Arduino Mega has more memory available, but due to limitations on how the Mega works, only some of the storage space is accessible by the driver code. Only around 50k is available for animation storage on a Mega.

An ESP32 has in comparison far more memory available for animation storage. Out of the box, you'll get around 1MB of space available (That could be an hour of animation data or more, depending on number

of keyframes). By changing the partition table on the ESP32 (Advanced but possible) you could access as much as 3mb of data on a standard ESP32.

Playing Exported Animations - Save To SD Card

If you want to save out your animation data and not worry about storage space at all, you can save it out to a format that can be read via an SD card. Note however the following limitations:

- 1) This currently only supports ESP32 based boards. Additional boards will be supported in the future, but an Uno R3 is unlikely to ever get this functionality.
- 2) You need to bring your own and hookup an SD card reader to your dev board. There's lots of tutorials out there on the internet, and you can get a maker friendly SD card reader module for a very low price.

In the Modules file, change:

```
// #define USE_SD_CARD_COMMAND_STREAM
```

To

```
#define USE_SD_CARD_COMMAND_STREAM
```

(Make sure you haven't also enabled the code version at the same time).

As well, in BottangoArduinoConfig.h you'll want to set the MISO, MOSI, Clock, and Chip Select pins to match how you've hooked up your SD card module:

```
#define SDPIN_CS 5           // chip select pin
#define SDPIN_MOSI 23         // MOSI pin
#define SDPIN_CLK 18          // clock pin
#define SDPIN_MISO 19         // MISO pin
```

You'll need an SD card formatted in FAT32. That usually means you need an SD card with a capacity of 32gb or LESS. Most OS will only format larger capacity SD cards in ExFAT format, but will format 32gb capacity or less in FAT32.

When you export to SD card animations, you'll get a folder named "anim." Copy that folder to the root of your SD card, and then insert the SD into your connected SD card module. Once you reupload the code with the above changes, you should be playing animation data from the SD card rather than over live USB.

Adding your own playback logic

When you export out you can set some code free playback options. You can set a starting clip, and idle clip, and different pin states to monitor in order to trigger certain animations with a button press, etc. However, you can certainly code your own logic to trigger exported animations however you'd like. Be sure to export out the animations with a configuration where there is no starting or idle animation, and then follow the next chapter to see where to place your own logic in the callbacks file.

-9-

Custom Events, Motors, and Other Callbacks

What is the Callbacks File?

Bottango provides callbacks in the callbacks file that allow you to input your own logic at various stages in the lifecycle of an effector or the driver itself.

In order to add your own logic, you'll modify the various methods in the "BottangoArduinoCallbacks.cpp" file. This chapter assumes you're comfortable writing basic C++ code.

If you are wanting to insert your own code or make otherwise unsupported changes to the Bottango driver, when in doubt, add the code to the callbacks file rather than trying to insert it in other source files or integrate the Bottango driver in whole into another set of functionality. It's a rare circumstance where what you need shouldn't be added to the Callbacks file rather than modifying the source.

Driver Lifecycle Events

Drivers offer the following four overall lifecycle events, for you to add your own logic to if required:

- **void onThisControllerStarted()**

Called after a successful handshake with the desktop app, but before effectors are registered. If you have effector specific startup needs, you should use the specific effector registered callback described below.

- **void onThisControllerStopped()**

Called after the driver receives a stop command and shuts down. The driver will stop all movement, deregister all effectors, and then call this callback.

- **void onEarlyLoop()**
- **void onLateLoop()**

These two loop callbacks provide easy access into the overall Arduino loop. Early loop happens before all effectors process their movement for this loop cycle, and late happens after. These are useful for if you have your own timing code you'd like to implement, independent of the Bottango animation timeline.

Custom exported animation playback logic

If you want to control exported animation playback logic, onEarlyLoop() is a great place to put it. Useful methods for controlling exported animations are:

```
// returns a bool for whether any exported animation is currently playing
bool isPlaying = BottangoCore::commandStreamProvider->streamIsInProgress()

// starts an exported animation by index and a bool for if should loop or not
// code export places an index key in the generated .cpp file
// sd card export has the name of the animation in the config file for each animation
BottangoCore::commandStreamProvider->startCommandStream(2, true);

// stops playing an exported animation if any is playing
BottangoCore::commandStreamProvider->stop();
```

Controlling the desktop app from your driver

There are a few commands available in the driver itself to communicate back to the desktop app and control it. In the BottangoArduinoCallbacks file, you can see examples of how to call each in the `onLateLoop` method.

You're able to control the following:

- Start / stop playing an animation from current animation and time
- Start / stop playing an animation with a given animation index and time
- Call "STOP" in Bottango (the same as pressing the escape key on the keyboard).
-

```
// EX: Request stop on driver, and disconnect all active connections
Outgoing::outgoing_requestEStop();

// EX: Pause Playing in App
Outgoing::outgoing_requestStopPlay();

// EX: Start Playing in App (in current animation and time)
Outgoing::outgoing_requestStartPlay();

// EX: Start Playing in App (with animation index, and start time in milliseconds) (-1 for index selects the current selected
// animation, -1 for time maintains the current time in app)
Outgoing::outgoing_requestStartPlay(1,1000);
```

Make sure to include the "src/Outgoing.h" header file in any other code file that would like to make these same calls.

What are Custom Motors and Events

The Bottango applications allows you to define and control custom events and motors. These represent hardware that don't fit nicely into the out of the box supported effectors that you might want to control.

If you add a custom event of any kind into the desktop app, you'll need to provide the functionality in the callbacks file of how to handle that event. Bottango will provide the what and the when of the animation,

you need to provide the how. In other words, Bottango will say custom motor X should go to signal Y at the right time, but you'll need to provide your own logic on how to actually express that desired signal.

Effector Identifier

Every motor registered has a unique eight character c-string identifier. In the bottango application, click on a motor to see its identifier. For default motors, the identifier is generated automatically from the pins used to control it, in combination with an i2c address if it exists. For custom motors and events, you define that identifier yourself as a custom c-string.

Each method dealing with a motor or effector in the "BottangoArduinoCallbacks.cpp" file has a pointer to an `AbstractEffector *effector` as one of the parameters in the method. In order to determine which motor is being acted on in a particular call to a method, you can access the identifier of the effector, and compare it against a desired identifier.

As an example, let's say you wanted to know if the call to a method was happening on a motor with the identifier "6"

```
char effectorIdentifier[9];                                // initialize a c-string to hold the identifier
effector->getIdentifer(effectorIdentifier, 9);           // fill the c-string with the identifier from the passed effector

if (strcmp(effectorIdentifier, "6") == 0)                  // strcmp(char* str1, char* str2) lets us know if the strings match
{
    // my logic here
}
```

Effector Lifecycle Events

Motors and effectors registered with a driver have the following lifecycle callbacks that allow you to input your own code

- **void onEffectorRegistered(AbstractEffector *effector)** - Called AFTER the motor is set live and registered with Bottango.

In this example, we turn on a light when an effector with the identifier 1 is registered.

```
void onEffectorRegistered(AbstractEffector *effector)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "1") == 0)
    {
        pinMode(LED_BUILTIN, OUTPUT);
        digitalWrite(LED_BUILTIN, HIGH);
    }
}
```

- **void onEffectorDeregistered(AbstractEffector *effector)** - Called BEFORE the motor is set NOT live and deregistered with Bottango.

In this example, we turn on a light when an effector with the identifier 1 is registered.

```
void onEffectorDeregistered(AbstractEffector *effector)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "1") == 0)
    {
        pinMode(LED_BUILTIN, OUTPUT);
        digitalWrite(LED_BUILTIN, LOW); }
}
```

- **void effectorSignalOnLoop(AbstractEffector *effector, int signal)** - Called every void loop() in the main loop thread, along with the int signal that effector is targeting in any current animation.

In this example, we turn on a light whenever an effector with the identifier 1 is greater than 1500.

```
void effectorSignalOnLoop(AbstractEffector *effector, int signal)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "1") == 0)
    {
        pinMode(LED_BUILTIN, OUTPUT);
        if (signal > 1500)
        {
            digitalWrite(LED_BUILTIN, HIGH);
        }
        else
        {
            digitalWrite(LED_BUILTIN, LOW);
        }
    }
}
```

Note that for stepper motors, the signalOnLoop function is called on each loop with the signal at the time the method is called and NOT on each step. This is because the steps themselves happen on an interrupt timer. It is essential that interrupt timer calls be as quick as possible, so it's not realistic to add additional callback logic to each stepper step, as even the pointer to the function takes precious cycles.

Auto Syncing Stepper Motors

You can initiate a stepper motor auto sync in the desktop app, in which case it will step continuously in the indicated direction. It's up to you to signal when that stepper has reached its home position. While a stepper is in an auto sync, after each step it will call back to isStepperAutoHomeComplete.

In this example, we return true, to signal that the stepper with a step pin on pin 6 is home, when pin 10 is read high (IE, a limit switch on that pin has been hit, etc).

```
bool isStepperAutoHomeComplete(AbstractEffector *effector)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "6") == 0)
    {
        pinMode(10, INPUT);
        if (digitalRead(10) == HIGH)
        {
            return true;
        }
    }
    return false;
}
```

Putting the Lifecycle Events Together for Custom Motors

In order to control a custom motor, you should have everything you need to drive your own motor types. You would use the register call back to initialize your motor, the deregister callback to shut down your motor, and the signalOnLoop callback to set its position based on Bottango's processing of animations.

In this example, we initialize, shut down, and set the position of a custom motor with the identifier "myMotor".

```
CustomMotor *myMotorInstance; // an instance of a class you have defined to control your motor type

void onEffectorRegistered(AbstractEffector *effector)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "myMotor") == 0)
    {
        myMotorInstance->setup(); // call your own logic to set up your motor
    }
}

void onEffectorDeregistered(AbstractEffector *effector)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "1 ") == 0)
    {
        myMotorInstance->shutDown(); // call your own logic to shut down your motor
    }
}

void effectorSignalOnLoop(AbstractEffector *effector, int signal)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "1 ") == 0)
    {
        myMotorInstance->setSignal(signal); // call your own logic to set your motor's position
    }
}
```

Responding to Custom Events

In the Bottango Application, you can create custom events of the following types:

- Curved (events with a range from 0 to 1)
- On / Off (events that are either on or off)
- Trigger (events that fire at particular times)
- Color (events that change color on a Red, Green and Blue scale)



In order to respond to those events, you'll need to input your own custom code. Note that unlike the signalOnLoop callback which happens every loop, the custom event callbacks happen ONLY when the signal changes.

In this example, we set the brightness of an LED with identifier "myLight" using a curved custom event.

Note: the float newMovement is a value from 0.0 to 1.0

```
void onCurvedCustomEventMovementChanged(AbstractEffector *effector, float newMovement)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "myLight") == 0)
    {
        pinMode(5, OUTPUT);
        int brightness = 255 * newMovement;
        analogWrite(5, brightness);
    }
}
```

In this example, we turn on and off an LED with identifier "myLight" using an on off custom event.

```
void onOnOffCustomEventOnOffChanged(AbstractEffector *effector, bool on)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "myLight") == 0)
    {
        pinMode(LED_BUILTIN, OUTPUT);
        digitalWrite(LED_BUILTIN, on ? HIGH : LOW); }
}
```

In this example, we set an LED's brightness to a random value with identifier "myLight" using a trigger custom event.

```
void onTriggerCustomEventTriggered(AbstractEffector *effector)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "myLight") == 0)
    {
        pinMode(5, OUTPUT);
        int brightness = random(0, 256);
        analogWrite(5, brightness); }
}
```

In this example, we set an RGB LED's color with identifier "myLight" using a color custom event.

```
void onColorCustomEventColorChanged(AbstractEffector *effector, byte newRed, byte newGreen, byte newBlue)
{
    char effectorIdentifier[9];
    effector->getIdentifier(effectorIdentifier, 9);

    if (strcmp(effectorIdentifier, "myRGB") == 0)
    {
        pinMode(3, OUTPUT);
        pinMode(5, OUTPUT);
        pinMode(6, OUTPUT);

        analogWrite(3, newRed);
        analogWrite(5, newGreen);
        analogWrite(6, newBlue); }
}
```

NOTE: More robust support for addressable RGB LED's like Neopixels is coming soon. In the meanwhile, join the Bottango discord to talk about what can be done now, and the limitations to work around to control Neopixels in Bottango currently.

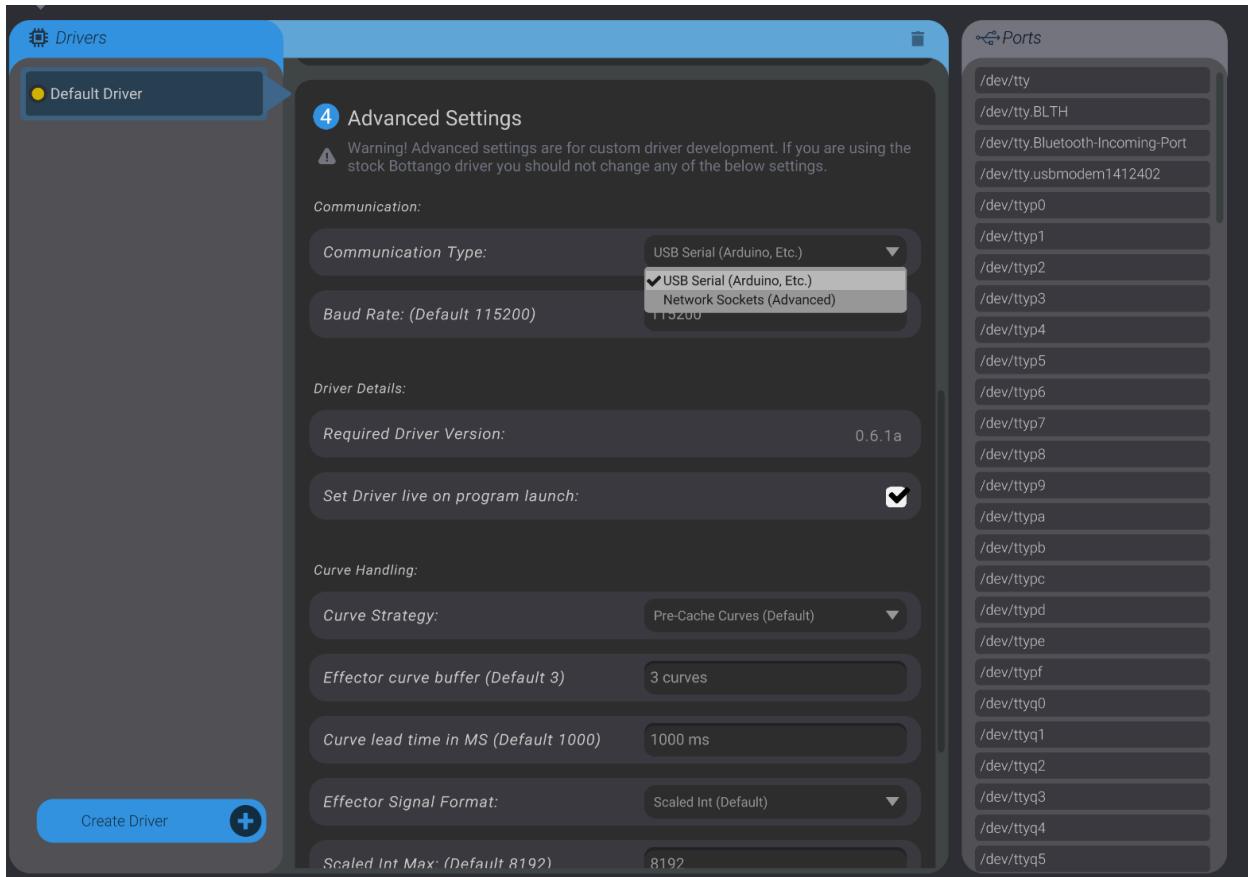
-7-

Using the Bottango Networked Driver

What is the Bottango Networked Driver?

The Bottango Networked Driver is an advanced tool for users comfortable writing their own code, and looking for flexibility beyond what is offered by USB serial connections or saving animations to baked code. It is made of two parts:

First, in Bottango you can set a hardware driver in the advanced settings to operate as a sockets based networked server, instead of communicating via USB serial.



Second, there is example code, written in Python, of a socket based client that can connect to the server and respond to commands from Bottango. You are not limited to using Python, that is just the fully functional example code.

When is the Bottango Networked Driver the right choice for me?

If your desired use case is more advanced than what can be done over USB serial or exported to code, then the most flexible alternative is the Bottango Networked Driver. Some example use cases where the Bottango Networked Driver makes sense:

- I want to communicate from the Bottango app to my own code running on a computer.
- I want to control a motor that requires a connection to a laptop/desktop computer instead of a microcontroller.
- I want to control a robot with Bottango wirelessly, and I'm able to use a Python compatible microcontroller, or can rewrite the C++ code to create and maintain a socket connection for streamed data (providing a networked variant of the C++ code is on the Bottango roadmap, but not yet delivered)

Starting a Bottango Network Server

In Bottango, as shown in the above image, you can set a hardware driver to operate as a sockets based networked server, instead of communicating via USB serial.

When you do so, and set that driver live, you will be creating a sockets based server on your network, at the port you indicated. That server will listen for incoming connections, and stream commands to the connected clients for you.

That server is located at the ip address of the computer running the Bottango application. Connecting to that server with a client on a local network should use the local network address of the computer, and the port you entered into Bottango (the default port is 59225). In order to communicate over the public facing internet, you would need to have the client connect to the public IP address of the computer, and you would be responsible for opening/forwarding the port as needed by your local network configuration.

Running a Network Client

The client that connects to the created server can be anything that can open and communicate in network connection via sockets. The API that communicates commands back and forth in the C++ USB serial microcontroller code and over the network is identical, and fully documented in the next chapter. You are welcome to build your own network client if needed, using the API documentation provided.

However, Bottango also provides a fully functional example implementation of a client that can connect to the network server in Python 3. If you're trying to just get up and going fast, the example Python code is fully functional and will be supported for future features.

Modifying the example Python Code

The most important file, if you want to quickly modify the Python code to meet your needs is the "CallbacksAndConfiguration.py" file, located in the following directory in the Bottango installation .zip downloaded from the Bottango website: Bottango/NetworkedDriver/src/CallbacksAndConfiguration.py

In this file you can set the address and port that you want to connect to. As well, there are callbacks for registering, deregistering, and setting signal on effectors, with documentation of what parameters are passed to those callbacks.

Make sure to set these configuration fields to match your server's setup:

```
address = '127.0.0.1'                                # The server's hostname or IP address
port = 59225                                         # The port used by the server
```

These other configuration fields are less likely to be needed:

```
log = True                                              # enable logging
roundSignalToInt = True                                 # treat signal as an int (true) or as a float (false)
apiVersion = "0.5.0b"                                   # api version to send in handshake response
```

Add your custom code as needed to the following lifecycle callbacks:

This callback is called whenever an effector is registered

```
handleEffectorRegistered(effectortype, identifier, minSignal, maxSignal, startingSignal);
```

This callback is called whenever an effector is deregistered

```
handleEffectorDeregistered(identifier);
```

This callback is called whenever an effector changes its target signal for any reason

```
handleEffectorSetSignal(effectortype, identifier, signal);
```

This callback is called whenever an on/off custom event changes its target on/off state for any reason

```
handleEffectorSetOnOff(effectortype, identifier, on);
```

This callback is called whenever an trigger custom event fires for any reason

```
handleEffectorSetTrigger(effectortype, identifier);
```