BOTTANGO ARDUINO DRIVER DOCUMENTATION

Documentation rev 10

-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-
Set Up an Arduino 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.

This documentation covers some pretty advanced topics that are not needed to get started with Bottango.

<u>You only need to read and follow this first chapter to get started!</u>

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

If you are using an Arduino Uno, 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. If you're just starting out, and have one of those boards, read the regular Bottango manual for instructions on uploading the driver to your Arduino using the desktop app.

Otherwise, if you are having trouble uploading the driver via the desktop app, you are using a board other than the three above, or want to make changes to the driver or it's configuration, read on!

What do you need

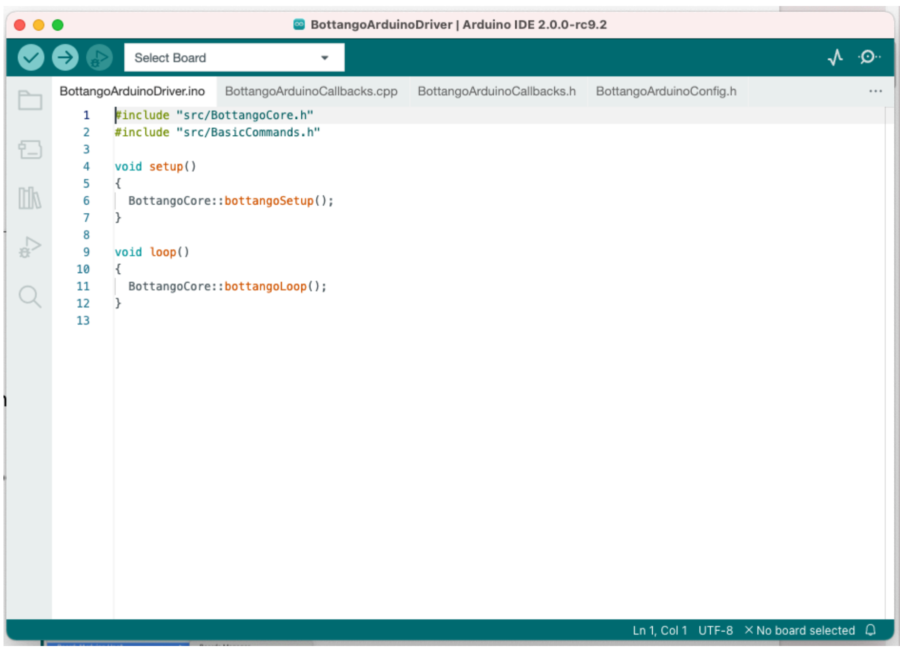
In order to set up your Arduino with Bottango using the driver source code, you will need the following:

- 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

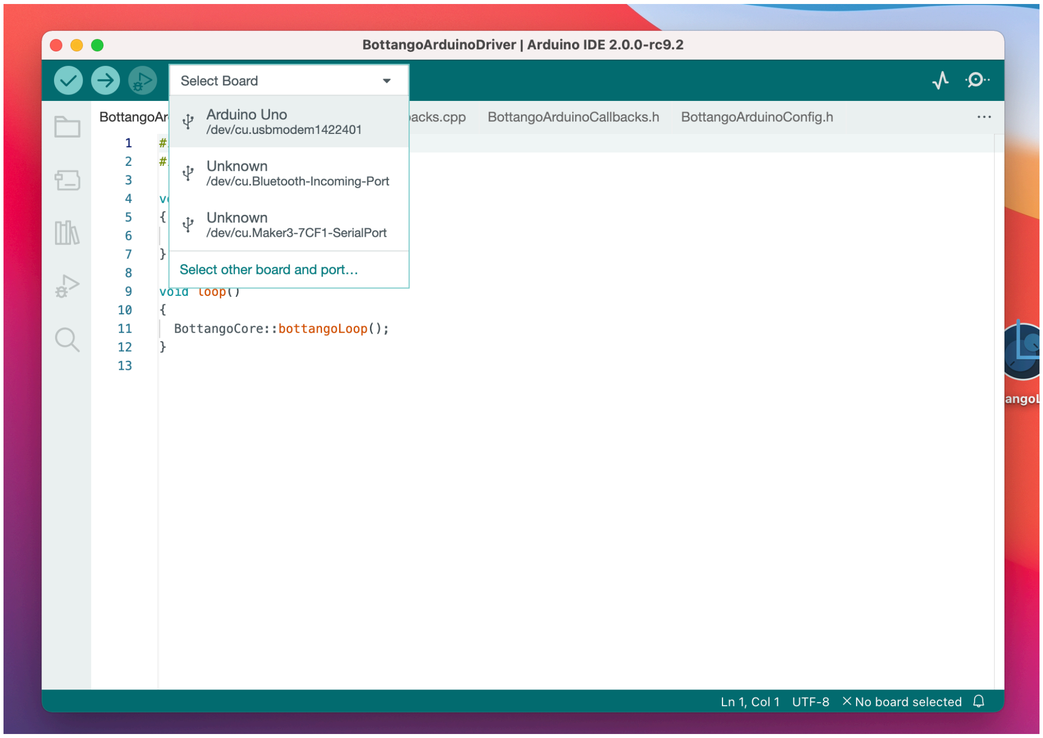
1 Open the BottangoArduino.ino file, which should open in the Arduino IDE if you have it installed.

You will see the BottangoArduino.ino file opened:



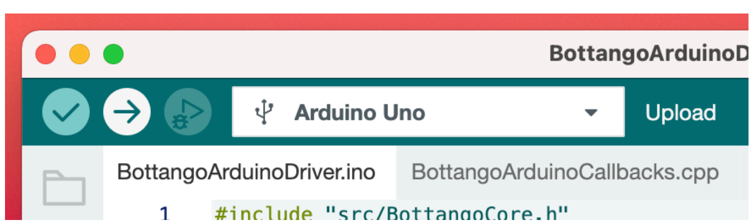
2 Connect your Arduino compatible microcontroller to your computer via USB.

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, or read on to see advanced usage of the Bottango Arduino code.

If you're just getting started, don't read the rest of the chapters in this documentation yet! It covers fairly advanced topics you won't need yet. Most all out of the box functionality is enabled just by uploading the provided code.

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

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 is optimized for and tested thoroughly on microcontrollers with ATMega328p and ATmega2560 chips. In practice, this means Arduino Uno, Mega and Nano boards. When in doubt, you'll find the most compatibility with those three types of Arduinos.

However, Bottango can in theory work on any Arduino compatible microcontroller! Bottango has been tested with ESP32, Arduino Zero, Teensy, and all kinds of other boards. For the most part, lots of boards work out of the box!

As of right now, servo support is spotty on the new Arduino Uno r4. This is not a Bottango specific issue, but with the servo libraries for the Arduino Uno r4. This is likely to be resolved in the future by the Arduino development team however.

-3-
Controlling Lots of Motors

The Limits of a Single Arduino

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 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.

Split Your Motors Across Multiple Arduinos

In order to get best results, there are optimizations you can take to get great performance for controlling lots of motors. But the core idea is this:

> 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!

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

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

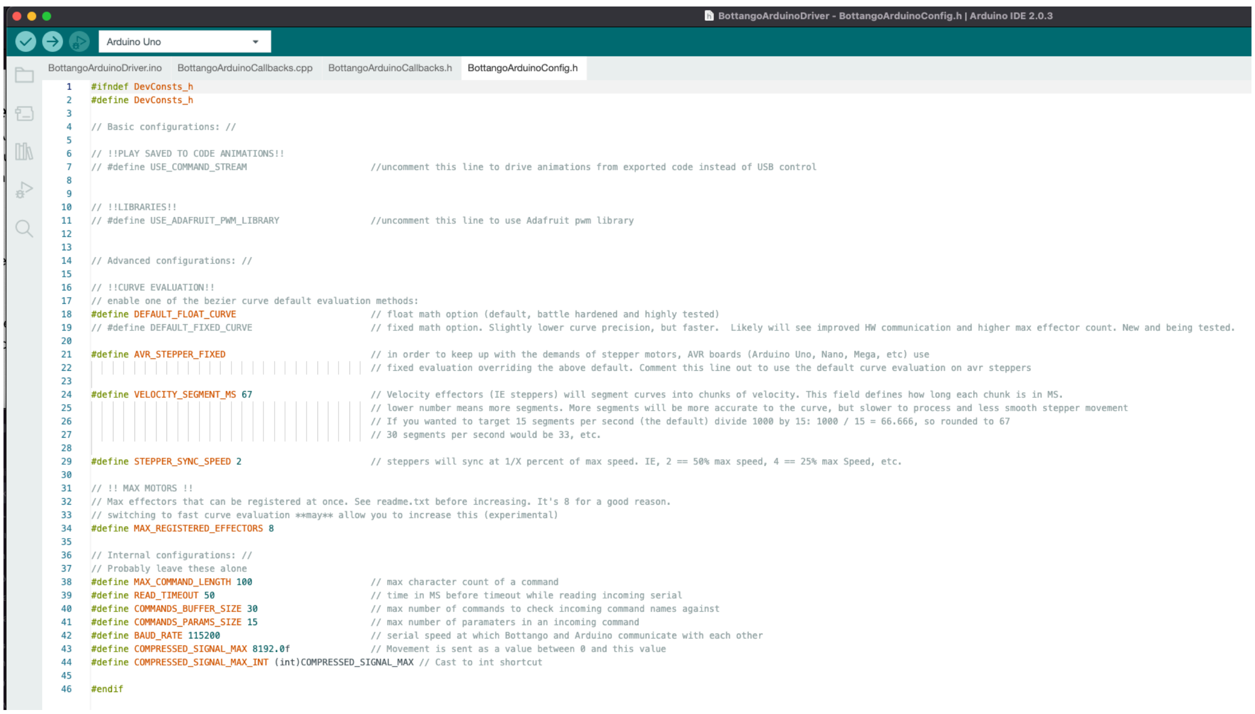
What happens if you get out of ideal and into the "Allowed" zone? You may see slightly more sluggishness or choppiness, depending on the complexity of the animations being sent to the Arduino (in terms of number of unique keyframes). Depending on your needs you might not see any difference, or it might be small enough to not matter. Stepper motors especially will get more choppy the more you add to the Arduino.

If you want to go past the maximum 8 motors per Arduino, see chapter 4 (Configuration File), but you should only do that if you know you're using a microcontroller able to keep up with the requirements of that many motors. An ESP32 is a great choice for a microcontroller that can handle more than 8 servos with Bottango.

-4-
The Configuration File

How To Edit the Configuration

Included in the Bottango Arduino code is a configuration file. This allows you to change some of the ways the Bottango Arduino code works. You can access it in the Arduino IDE by opening the BottangoArduino.ino file, and then clicking on the tab labeled "BottangoArduinoConfig.h"




```
#ifndef DevConsts_h
#define DevConsts_h

// Basic configurations: //

// !!PLAY SAVED TO CODE ANIMATIONS!!
// #define USE_COMMAND_STREAM                   //uncomment this line to drive animations from exported code instead of USB control


// !!LIBRARIES!!
// #define USE_ADAFRUIT_PWM_LIBRARY             //uncomment this line to use Adafruit pwm library


// Advanced configurations: //

// !!CURVE EVALUATION!!
// enable one of the bezier curve default evaluation methods:
#define DEFAULT_FLOAT_CURVE                     // float math option (default, battle hardened and highly tested)
// #define DEFAULT_FIXED_CURVE                  // fixed math option. Slightly lower curve precision, but faster.  Likely will see improved HW communication and higher max effector count. New and being tested.

#define AVR_STEPPER_FIXED                       // in order to keep up with the demands of stepper motors, AVR boards (Arduino Uno, Nano, Mega, etc) use
                                                // fixed evaluation overriding the above default. Comment this line out to use the default curve evaluation on avr steppers

#define VELOCITY_SEGMENT_MS 67                  // Velocity effectors (IE steppers) will segment curves into chunks of velocity. This field defines how long each chunk is in MS.
                                                // lower number means more segments. More segments will be more accurate to the curve, but slower to process and less smooth stepper movement
                                                // If you wanted to target 15 segments per second (the default) divide 1000 by 15: 1000 / 15 = 66.666, so rounded to 67
                                                // 30 segments per second would be 33, etc.

#define STEPPER_SYNC_SPEED 2                    // steppers will sync at 1/X percent of max speed. IE, 2 == 50% max speed, 4 == 25% max Speed, etc.

// !! MAX MOTORS !!
// Max effectors that can be registered at once. See readme.txt before increasing. It's 8 for a good reason.
// switching to fast curve evaluation **may** allow you to increase this (experimental)
#define MAX_REGISTERED_EFFECTORS 8

// Internal configurations: //
// Probably leave these alone
#define MAX_COMMAND_LENGTH 100                  // max character count of a command
#define READ_TIMEOUT 50                         // time in MS before timeout while reading incoming serial
#define COMMANDS_BUFFER_SIZE 30                 // max number of commands to check incoming command names against
#define COMMANDS_PARAMS_SIZE 15                 // max number of paramaters in an incoming command
#define BAUD_RATE 115200                        // serial speed at which Bottango and Arduino communicate with each other
#define COMPRESSED_SIGNAL_MAX 8192.0f           // Movement is sent as a value between 0 and this value
#define COMPRESSED_SIGNAL_MAX_INT (int)COMPRESSED_SIGNAL_MAX // Cast to int shortcut

#endif
```

Below is the description of the most commonly changed configurations. If it's not described here, you should probably leave it alone.

Changing max motors

"#define MAX_REGISTERED_EFFECTORS 8"

If you wanted to change the maximum number of motors on a board, you would change that number from 8 to some other number, but as stated in the previous chapter, this is best done only if you fully understand what would be needed to support that change.

Enabling Saved Animations

"// #define USE_COMMAND_STREAM "

When you save out animations, you need to let the Bottango Arduino code know to use saved animations instead of listening for animations to come over USB.

In the line "// #define USE_COMMAND_STREAM " remove the two slashes at the beginning of the line ("//") to enable that functionality. You'll also need to include the "GeneratedCommandStream" files that the Bottango application generates for you on exporting animations in the same folder as the BottangoArduino.ino file.

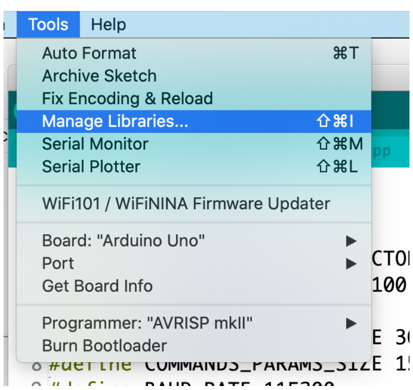
Save the file, and re-upload the code to your microcontroller.

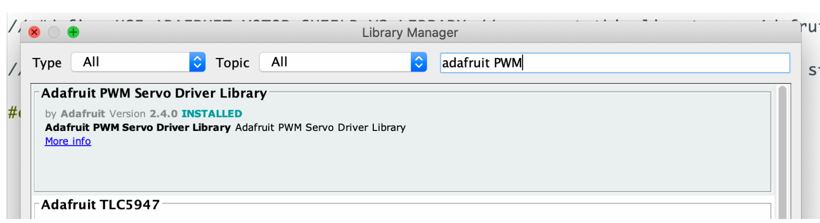
Using Adafruit 16 Channel I2C Servo Library

"// #define USE_ADAFRUIT_PWM_LIBRARY "

You can control servos in Bottango with the Adafruit 16 channel PWM shield, and steppers with the Adafruit motor shield v2. In order to do so you must first install the corresponding Adafruit library on your computer. You must install the libraries using the Arduino IDE library manager, so that the libraries will be installed in the default location.

Select Tools > Manage Libraries... and then find the corresponding library and install it.





In the line "// #define USE_ADAFRUIT_PWM_LIBRARY " remove the two slashes at the beginning of the line ("//") to enable support for the 16 channel PWM shield.

Save the file, and re-upload the code to your microcontroller.

NOTE: The Adafruit 16 channel PWM shield introduces a layer of communication from the Arduino to the PWM Shield which takes additional time and processing.

Because of this extra communication step, you'll see sluggishness and slow-downs more often when using the 16 channel PWM shield vs using a direct pin connection.

For this reason, as well as because of the 8 servo limit, I recommend against using the Adafruit 16 channel PWM shield vs direct pin control with Bottango if you can avoid it.

Curve Evaluation type

"#define DEFAULT_FLOAT_CURVE"
"// #define DEFAULT_FIXED_CURVE"

By default Bottango uses floating point math to evaluate animations. This is tried and true, and has had years of testing against it. However… it's also slow, and one of the main reasons for the 8 motor max per board. Bottango has recently added a fixed point math version of animation evaluation which is faster. If you'd like to try it out, and help test, switch from default float to default fixed by commenting out the float line and removing the "//" on the fixed line.

Switching to fixed animation evaluation may allow you to increase the maximum number of motors, see better performance on the i2c pwm shield etc. But it also is new and still in testing, so proceed with caution!

Stepper Behavior

"#define VELOCITY_SEGMENT_MS 67"

Steppers segment animation curves into series of velocities, in order to reduce the calculation time and increase the smoothness of the output. This field is how long each segment is in MS.

If you wanted to target 15 segments per second (the default), you would divide 1000 by 15, IE 66.666, which rounds up to 67.

Higher numbers means less segments: Less jitter at the cost of lower accuracy to the curve of the animation

Lower numbers mean more segments: More accurate to the curve, but less smooth.

"#define STEPPER_SYNC_SPEED 2"

This controls the speed of a stepper when syncing. Steppers will sync at 1 / X as a percent of the maximum speed of the stepper. By default, that's 1 / 2, IE 50% of the max speed.

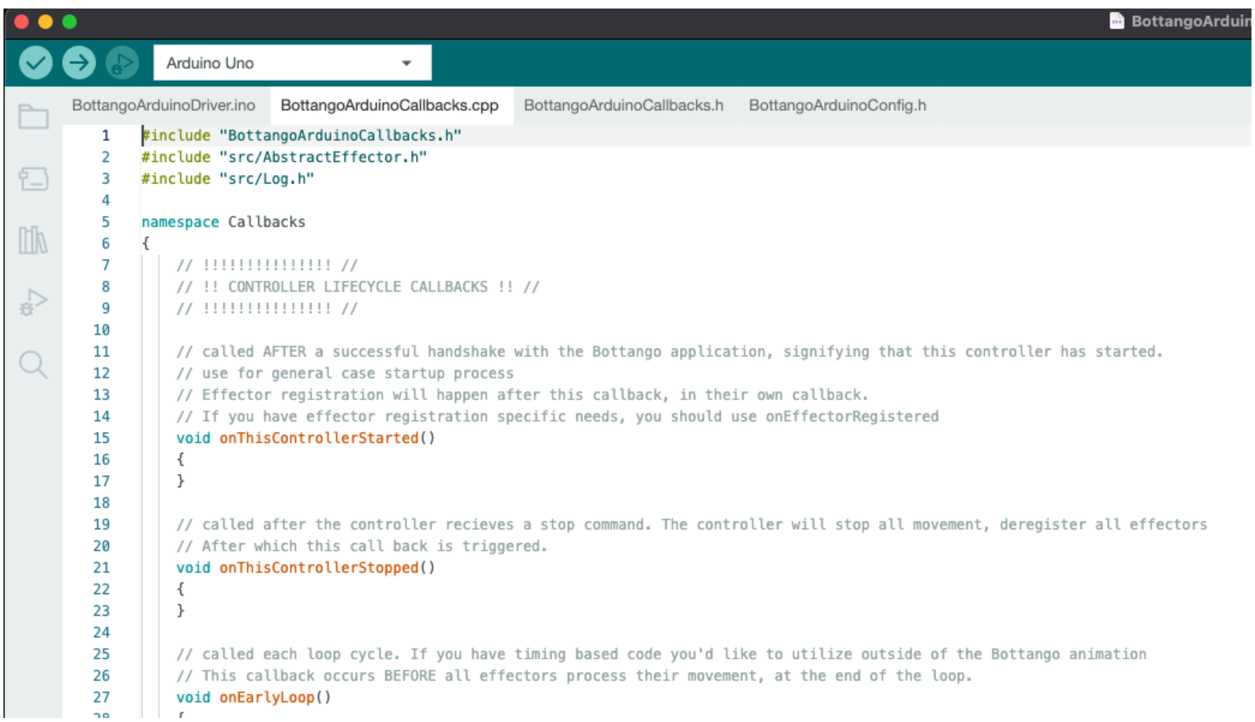
-5-
Custom Events, Motors, and Lifecycle Callbacks

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.

As well, Bottango provides callbacks, that allow you to input your own logic at various stages in the lifecycle of an effector.

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.



Effector Identifier

Every motor registered with an Arduino has a unique eight character c-string identifier. In the bottango application, click on a motor to see it's identifier. For default motors, the identifier is generated 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.

Each method 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 and 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->getIdentifier(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
}
```

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.

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.

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.

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.

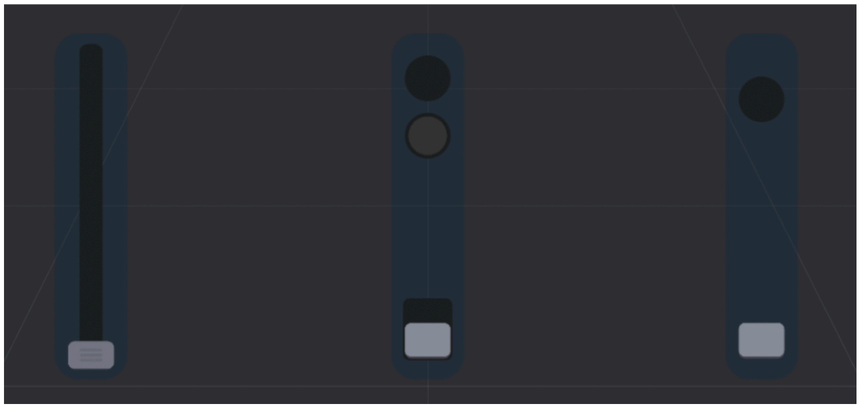
In this example, we initialize, shut down, and set the position of a custom motor with the identifier "myMotor".

```
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);
    }
}
```

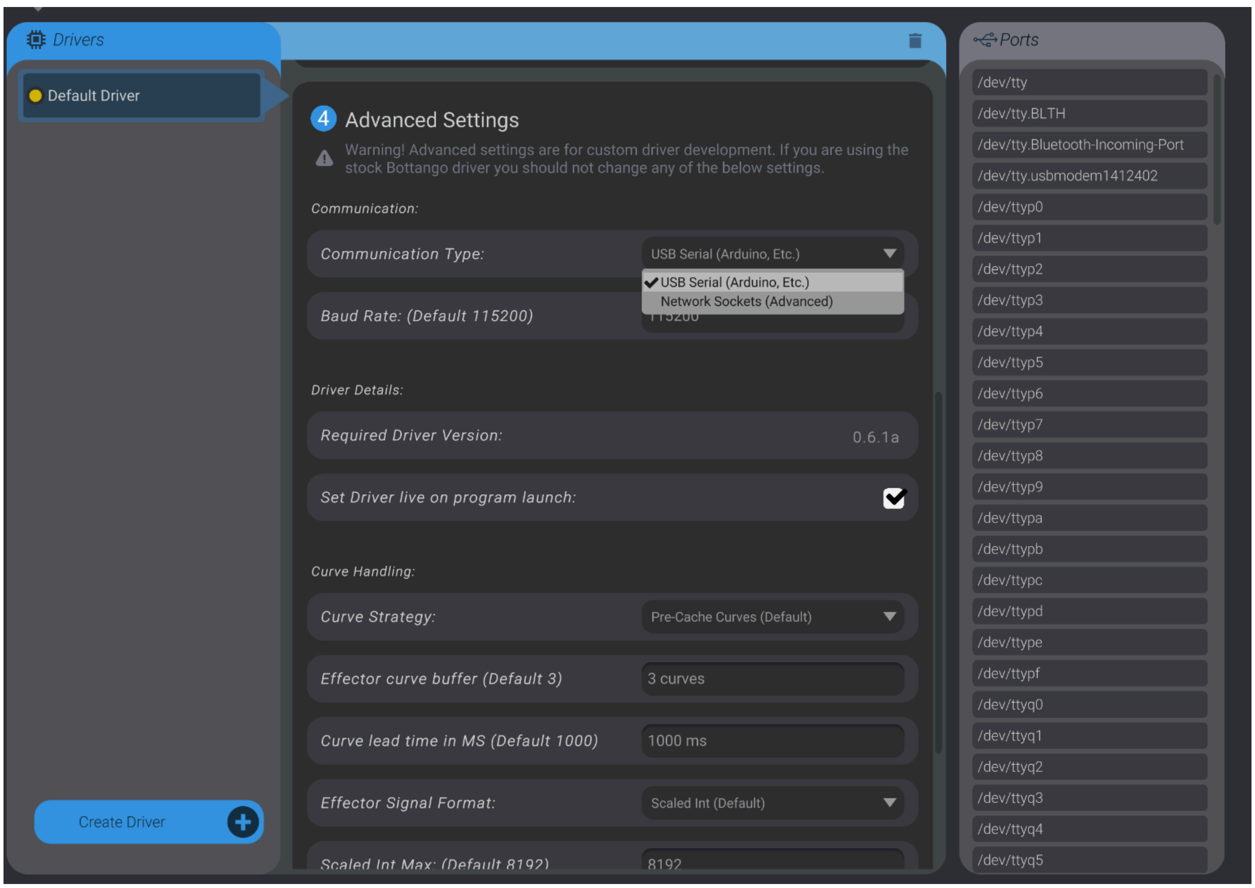
NOTE: More robust support for addressable RGB LED's like Neopixels is coming soon. In the meanwhile, join the Bottango discord to talk about what can be done now, and the limitations to work around to control Neopixels in Bottango currently.

-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 is 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):