BOTTANGO DRIVER API

Documentation rev 4

-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

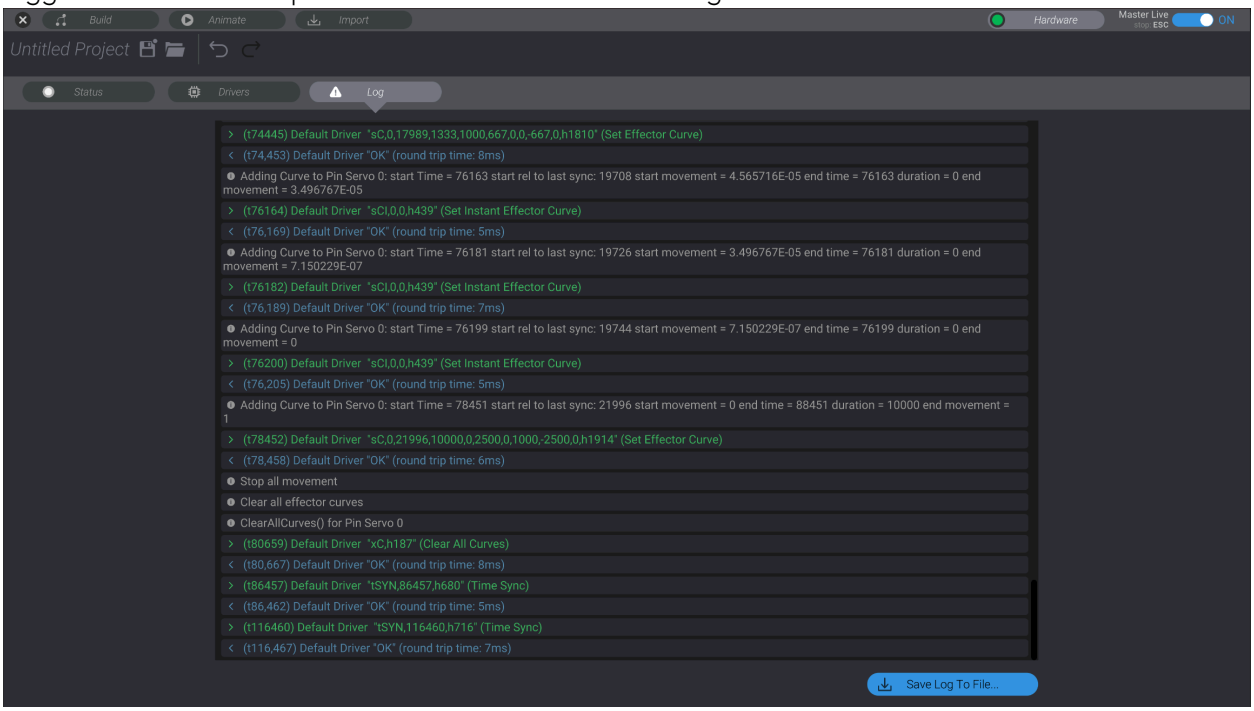
-2-
Bottango Driver API

Create Your Own Bottango Microcontroller Code

Bottango provides the included open source Arduino compatible code so you can communicate with the Bottango application using an Arduino compatible microcontroller. Bottango as well provides an example Python implementation of the driver code as well for networked communication. If you were so inclined, you could implement your own microcontroller or networked code to respond to the Bottango driver API on any kind of microcontroller or network capable machine.

If you do so, please let us know! We'd love to see what you come up with!

You can monitor all incoming and outgoing serial commands in the Bottango application using the logger. This can be a helpful tool to understand how Bottango works behind the scenes.



The Bottango Driver API

Commands are terminated with a '\n' character. Parameters are separates with a ',' character.

Every command contains a hash code as the final parameter. This allows you to sanity check that the command string did not get errors introduced over the line. If you would like to use the hash code, see the method bool checkHash(char *cmdString) in the C++ file src/BottangoCore.cpp to see the hash

check logic. Hash parameters start with an 'h' character. As an example, you might see the command "xUC,4,h368"

The Python network driver ignores the hash value. Error rates in network communication are far lower than over serial, as network communication contains much more redundancy and error checking inherently.

As a reminder again, commands are terminated with a '\n' character. Parameters are separates with a ',' character.

— BASIC LIFECYCLE —

The lifecycle of communication begins with a handshake request from the Bottango application, which should elicit a handshake response from the microcontroller.

• *Outgoing Arduino startup, on serial opened*
<- BOOT

• *Incoming Handshake Request after BOOT is received:*
Param 0: random code for verification
-> hRQ,144

• *Outgoing Handshake Response:*
Param 0: current driver version
Param 1: repeat back random code for verification
Param 2: bool (as int) true if accepting commands, false in saved animation playback mode
<- btngoHSK,0.3.5c,144,1

• *Incoming Time Sync*
(see src/time.cpp for how Bottango keeps time between the Application and the microcontroller in sync)
Param 0, time in MS since LAST time sync.
-> tSYN, 30000

— START AND STOP —

• *Incoming Deregister All Effectors Currently registered*
-> xE

• *Incoming Deregister a specific effector*
Param 0, identifier of effector
-> xUE,3

• *Incoming clear all curves currently playing on all registered effectors*
-> xC

• *Incoming clear all curves currently playing on a specific effector*

-> xUC,3

— REGISTER EFFECTOR —

Note: There is no update an effector. If a change occurs, the effector is deregistered, and then a new replacement with updated data is registered.

• *Incoming register a servo with pin control*
Param 0, pin
Param 1, minimum PWM signal
Param 2, maximum PWM signal
Param 3, maximum PWM signal change per second
Param 4, starting PWM signal
-> rSVPin,3,800,1900,600,1500

• *Incoming register a servo with i2c control*
Param 0, i2c address
Param 1, pin
Param 2, minimum PWM signal
Param 3, maximum PWM signal
Param 4, maximum PWM signal change per second
Param 5, starting PWM signal
-> rSVPin,64,3,800,1900,600,1500

• *Incoming register a stepper with 4 pin control*
Param 0, pin 0
Param 1, pin 1
Param 2, pin 2
Param 3, pin 3
Param 4, maximum counter clockwise steps from home (will always be negative)
Param 5, maximum clockwise steps from home (will always be positive)
Param 6, starting steps offsets from home (negative or positive)
-> rSTPin,3,4,5,6,-500,500,0

• *Incoming register a stepper with step dir control*
Param 0, step Pin
Param 1, dir Pin
Param 2, bool (as int), true if clockwise on low, false if counterclockwise on low
Param 3, maximum counter clockwise steps from home (will always be negative)
Param 4, maximum clockwise steps from home (will always be positive)
Param 5, starting steps offsets from home (negative or positive)
-> rSTDir,3,4,1,-500,500,0

• *Incoming register a stepper with i2c control*

Param 0, i2c address
Param 1, pin
Param 2, maximum counter clockwise steps from home (will always be negative)
Param 3, maximum clockwise steps from home (will always be positive)
Param 4, starting steps offsets from home (negative or positive)
-> rSTPin,64,0,-500,500,0

• *Incoming register a curved custom event*
Param 0, identifier
Param 1, max movement per second
Param 2, starting movement
-> rECC,myEvent,0.25,0.5

• *Incoming register an on/off custom event*
Param 0, identifier
Param 1, bool as int, true if should start on.
-> rECOnOff,0

• *Incoming register a trigger custom event*
Param 0, identifier
Param 1, bool as int, true if should start on.
-> rECTrig,Off,0

• *Incoming register a color custom event*
Param 0, identifier
Param 1, integer between 0 - 255 for red channel of starting color.
Param 2, integer between 0 - 255 for green channel of starting color.
Param 3, integer between 0 - 255 for blue channel of starting color.
-> rECColor,255,0,255

• *Incoming register a custom motor*
Param 0, identifier
Param 1, minimum signal
Param 2, maximum signal
Param 3, maximum signal change per second
Param 4, starting signal
-> rMTR,myMotor,-1000,1000,300,0

• *Incoming manually sync an effectors signal in order to home it*
Param 0, identifier
Param 1, amount of signal to sync by (can be negative or positive)
-> sycM,myMotor,-100

— ANIMATION CURVES —

After sending an animation curve, Bottango will wait until it gets the "ready for next" signal before sending the next one, in order to prevent serial buffer overrun

• Outgoing Ready For Next <- \nOK\n

Bottango attempts to cache curves on the microcontroller, rather than send them just in time, whenever possible. When a curve is 1 second or less away from being played, Bottango will send the curve to the microcontroller, unless at least 3 curves are currently cached and waiting, in which case it will send the most immediate three, and wait to send the next curve until Bottango predicts that a cached curve has been played and can be removed.

Bottango sends to the microcontroller animation curves, rather than sampled data. This takes the form of a starting and ending keyframe and duration.

Please see src/BezierCurve.cpp to see how the incoming curve is translated into animation and sampled over time.

• *Incoming Set Motor Curve*
Param 0, identifier
Param 1, time of start in MS relative to last sync time (could be positive or negative)
Param 2, duration of curve in MS
Param 3, start movement (expressed as an int between 0 - 1000)
Param 4, start control point tangent X (relative to start time in MS)
Param 5, start control point tangent Y (relative to start movement) (expressed as an int between 0 - 1000)
Param 6, end movement (expressed as an int between 0 - 1000)
Param 7, end control point tangent X (relative to start time in MS) (this will be a negative number or 0)
Param 8, end control point tangent Y (relative to start movement) (expressed as an int between 0 - 1000)
-> sC,3,2029,1500,250,0,500,750,-250,0

• *Incoming Set On Off Curve*
Param 0, identifier,
Param 1, time of start in MS relative to last sync time
Param 2, bool as int, true if on
-> sCO,myEvent,322,1

• *Incoming Set Trigger Curve*
Param 0, identifier,
Param 1, time of start in MS relative to last sync time
-> sCT,myEvent,322

• *Incoming Set Color Curve*
Param 0, identifier,
Param 1, time of start in MS relative to last sync time

Param 2, duration of curve in MS
Param 3, red channel of starting color of curve (int between 0 - 255)
Param 4, green channel of starting color of curve (int between 0 - 255)
Param 5, blue channel of starting color of curve (int between 0 - 255)
Param 6, red channel of ending color of curve (int between 0 - 255)
Param 7, green channel of ending color of curve (int between 0 - 255)
Param 8, blue channel of ending color of curve (int between 0 - 255)
-> sCC,myEvent,322,1000,255,0,0,0,0,255
!Note that color curves are interpolated linearly!

If the curve that Bottango would send has a duration at or very close to instant and should be executed now, an instant curve is sent instead. This optimizes the size of the serial stream, and allows for more responsive streaming of commands. Behind the scenes, the Bottango Arduino code turns these instant curves into zero length regular curves with a start time at the time the command is received. This is purely a serial size space saving command.

• *Incoming Set Instant Curve*
Param 0, identifier
Param 1, target movement (expressed as an int between 0 - 1000)
-> sCI,3,550

• *Incoming Set Instant Color Curve*
Param 0, identifier
Param 1, red channel of color to be instantly set (int between 0 - 255)
Param 2, green channel of color to be instantly set (int between 0 - 255)
Param 3, blue channel of color to be instantly set (int between 0 - 255)
-> sCCI,myEvent,255,0,0