About:
"csStateBheavior" is a micro-controller program for running state-based behavioral tasks and/or stimulus delivery.

Dependencies:
csStateBehavior uses some 3rd party libraries. Becuause the program is embedded, you need these libraries to compile and upload the code to a Teensy, even if you aren't going to use the features they offer. You can install libraries into the Arduino IDE in a variety of ways, but I recommend downloading these libraries, placing them in a .zip and installing them by selecting "Sketch -> Include Library -> Add .zip library"

A) Adafruit's neopixel library:
https://github.com/adafruit/Adafruit_NeoPixel
https://github.com/adafruit/Adafruit_NeoPixel/archive/master.zip
This is used to control neopixel strips. Intended use is to have neopixel strips like this: https://www.adafruit.com/product/2847 or any of them, in a behavior rig enabling you to change light colors for cues, habituation reasons, or just to have light around.

B) HX711 ADC Library by Bodge:
https://github.com/bogde/HX711
https://github.com/bogde/HX711/archive/master.zip
This is to interface with and read data from the loadcell. There are a ton of HX711 libraries around, but Bodge got all the "little things" right.

C) Adafruit's MCP4725 (DAC) Library
https://github.com/adafruit/Adafruit_MCP4725
https://github.com/adafruit/Adafruit_MCP4725/archive/master.zip
There are a bunch of MCP libraries around too. This one is fine, and I like Adafruit. This is used to add two more DAC channels via MCP4725 breakouts. I use one from Adafruit and one from Sparkfun, because the default addresses differ. But, you can change the address of either and just buy two from one place.
https://www.adafruit.com/product/935

Basics:
By default, csStateBehavior runs at 1 kHz. The program is a state machine instantiated as a function that is timed using and interrupt timer. Each interrupt a function called *vStates* runs and it:
a) evaluates which state it should be in
b) runs a state header if it enters a new state
c) executes code determined by its state
d) streams, over serial, a data report
e) interacts with experimental devices, with precise timing, via digital, analog, i2c, spi, or UART IO.

It is important to note that the intended approach of the broader *csBehavior* library to stimulus and behavior control involves using a python program to interact with the teensy, such that real-time processes are effected by the Teensy and the python program saves data, shows plots, and makes determinations about when to change states. In this way, the Teensy program is designed to be as "dumb" as possible, yet it is the core of the behavioral control stack. *csStateBehavior* provides the master clock, and the state the Teensy is in, is considered by all other interacting processes (like the python program) to be "ground truth." Programs can request state changes, but the teensy changes the state and provides feedback when it has.

csStateBehavior, by default, sits in "*State 0; S0*" where it does very little. There is no data report sent in *S0*. However, in S0, the Teensy does evaluate whether it should move to a new state, responds to serial-based commands, and it can execute a series of pre-programmed device interactions "relays" based on specific relay pins being turned high. These relays are intended to allow other users the ability to gain access to devices attached to your Teensy/Behavior Rig without having to use the program or worry about weird current/voltage states on the dedicated trigger pins. For example, a user may be using Matlab code and NI-

DAQ boards to do their work, but they want to trigger a microscope, change light brightness, and trigger rewards from a pump. Instead of having to split connections, which can lead to unexpected behaviors, the user can trigger a pin on the Teensy and it will do any, or all, of those things. Also, in S0, the state of connected devices can be polled asynchronously for general use. For example, you can change the color of neopixel light strips, play tones, and check the values of attached sensors and devices. Lastly, certain sensors that cannot be polled in real-time can be used in S0 (such as environmental sensors).

Data Report:
What is streamed in the data report changes frequently, and can be customized. As of this writing, the data report streams the following in a serial packet that has a header called "tData:"

a) intCount: this is the actual number of interrupts fired since the Teensy has left S0. By default, csStateBehavior runs a 1kHz, thus, each interrupt is 1 ms. This should be considered to be the actual clock.

b) sesTime: this is the session clock, as determined by evaluating elapsedMillis(). Conceptually, this value should be identical to intCount. It can differ by 1 at the beginning, as sometimes the first ellapsedMillis will report 0 or 1. Beyond that, the two clocks should be identical. If it isn't, that means the function did not finish evaluating before the next interrupt. Despite common conception, Teensy's and related boards are very fast, each evaluation of vStates() takes less than 200 us.

c) stateTime: this is the state clock, as determined by evaluating elapsedMillis(), and resetting the reference each time you enter a new state. The intention of this variable is to help make evaluation of state-specific conditions/functions easier. Also, it may help with post-hoc analysis.

d) currentTeensyState (aka knownValues[0]): this is the current state the Teensy is in.

e) loadCellValue (aka knownValues[17]): this is the reading of the loadCell from the HX711 amplifier.

f) lickSensorA_Value: this is the value of the sensor used for determining licks. By default, even digital sensors are treated as analog allowing an external program to threshold as you see fit. csStateBehavior (the hardware) does not evaluate licks in real-time, and does not need to. Though, it is trivial to configure csStateBehavior to do hardware lick detection.

g) encoderAngle: this is the value reported by a rotary encoder used to track the location of a wheel.

h) pulseCountA: this is the incremental pulse count determined by an interrupt-based pulse counter. It can be used to keep track of elapsed frames from a two-photon microscope or ccd camera, etc. It resets every session.

i) loopTime: this is the amount of time in micro-seconds that the function took to evaluate for the last interrupt. The default clock rate of csStateBehavior is 1kHz, meaning that the interrupt timing the state machine function fires every ms, no exception. However, if the function does not return within a ms, you will fail to log those interrupts. Loop time allows you to identify problematic interrupts and fix any issue that may be slowing down the program. As configured, each loop of the main function takes 500-720 microseconds, depending on details of how it was compiled, primarily.

j-m) genAnalogInput0-3: these are the values of 4 different analog inputs chosen by the user. Just add your device to these and it will record the value. Note that the teensy is an arm-based computer and operates at 3.3V, the analog input has to be between 0 and 3.3V, not only to be precise, but the pins will become damaged if you drive them above 3.3V. Use a voltage divider to scale the peak voltage to 3.3V if you have something you know to exceed that voltage.

Generic State:
The generic state consists of the following:
A) State Header: code that only runs when you enter the state.
B) State Body: code that loops until you leave the state.

Every state runs the default state header and body. You can override these, and you can supplement them (prefered).

Anatomy of the default header:
The concept of “the header” is code you want to run only once and specifically when you arrive to the state. The obvious use of this is to reset a timer. Here is what the default header looks like:

```
void genericHeader(int stateNum) {
    // a: reset header timer
    headerTime = 0;
    // b: reset header states and set current state's header to 1 (fired).
    resetHeaders();
    headerStates[stateNum] = 1;
    // c: set analog output values to 0.
    analogOutVals[0] = 0;
    analogOutVals[1] = 0;
    analogOutVals[2] = 0;
    analogOutVals[3] = 0;
    // d: reset state timer.
    stateTime = 0;
}
```

What this does is to reset a header timer, resets the state of all other state’s headers to 0 (hasn’t fired) while also setting the current state’s header to 1 (has fired), then it zeros out the analog output voltages ensuring that if you entered the state in the middle of a stimulus, it doesn’t hang, and lastly it resets the state timer.

The concept of the state timer (stateTime) is most important in creating your own states. Typically, you will want to do things based on how long you have been in a state. You can program hardware specific logic, but I recommend using the python component to react to time events etc. I mention this here because this variable (stateTime) is streamed over to python.

Anatomy of the default state body:

```
void genericStateBody() {
    lickSensorAValue = analogRead(lickPinA);
    lickSensorAValue = analogRead(lickPinB);
    genAnalogInput0 = analogRead(genA0);
    genAnalogInput1 = analogRead(genA1);
    genAnalogInput2 = analogRead(genA2);
    genAnalogInput3 = analogRead(genA3);
    analogAngle = analogRead(analogMotion);
    writeAnalogOutValues(analogOutVals);
    If (scale.is_ready()) {
      scaleVal = scale.get_units() * 22000;
      // this scale factor gives hundreths of a gram as the least significant int
      knownValues[14] = scaleVal;
  }
}
```

What this does is, reads the values of up to two connected lick/touch sensors, the four generic analog inputs, an analog encoder, writes the currently desired/computed analog out voltages (see below) on the analog output channels, then it looks to see if load cell data is available, if so it grabs it and scales it to give useable numbers.

You can add specific code to the header or body in your custom state. Here is an example state:

```
// *************************
// State 1: Boot/Init State
// *************************
if (knownValues[0] == 1) {

  if (headerStates[1] == 0) {
    visStim(0);
    genericHeader(1);
    blockStateChange = 0;
   }
   genericStateBody();
 }
```

From the top, this state is entered if the state variable (knownValues[0]) is 1. If it enters the first time (as indicated by it’s header bit being 0) it will zero out a visual stimulus (added code), run the default header, and will allow python to control the state (blockStateChange = 0). After this, the header bit will be set to 1 (which happens in the default header) then, it can’t run anymore as long as the program is stuck in state 1. But, instead, it will always fire “genericStateBody()” that is the default state body. If I wanted to add some code that changed the state after 100 ms, I could add the following:

```
// *************************
// State 1: Boot/Init State
// *************************
if (knownValues[0] == 1) {

  if (headerStates[1] == 0) {
    visStim(0);
    genericHeader(1);
    blockStateChange = 0;
   }
   genericStateBody();
   if (stateTime >= 100){
     // note that stateTime is in milliseconds
     knownValues[0] = 2;
     // send teensy to state 2
   }
 }
```

But, as you will see in later parts of the documentation, it is recommended you handle such things in python. However, the more event logic you program on the teensy, the more close to true real-time performance you will get. The above code will move the teensy to state 2, in real-time, once 100 ms has elapsed, but if you are coordinating with the python component, it will need to be made aware.

There are two basic approaches to two components interacting dynamically, one is to have all events on the hardware, and stream the data for logging in software. Or, one can have the software log data, and determine state flow. The csBehavior library biases things towards software control in order to take advantage of richer computation and to allow maximal flexibility.

Analog Output:

Each sample's analog output value is kept in an array called "*analogOutVals*." By default, the generic state header, which every state executes, writes all DAC values to 0. *analogOutVals* is isolated from the array *pulseTrainVars*, which keeps track of all variables needed to generate waveforms. Thus, overriding the output value is non-destructive. If this doesn't seem important to you right now, that is ok there will be more on why this is important later. What matters for now is that *pulseTrainVars* and *analogOutVals* work with each other.

When you enter a state, the generic header will set the *analogOutVals* to 0. If you want to write a simple DC value on the DAC lines you can, even in the header. Each state's header allows for custom header code, that is executed after the genetic header. Thus, if you want two channels to be 3.3V for the whole state, you can set *analogOutVals[0] = 4095;* and *analogOutVals[1] = 4095;* in your header code. The reason we write the values to 0 by default is so the devices don't hang and so you can start fresh each state. Another benefit is that if you don't use analog out for a state, or at all, you don't have to do anything because the DAC lines will all write 0.

If you want controlled output in a state the generic pattern is to execute the following helper functions in the body:

stimGen(*pulseTrainVars*);
setAnalogOutValues(*analogOutVals,pulseTrainVars*);
writeAnalogOutValues(*analogOutVals*);

What these do is first, "stimGen" use all the variables in pulseTrainVars (explained later) to determine what value the channel should have based on a desired waveform and the current time. Second, "setAnalogOutValues" simply writes the calculated value from stimGen (stored in the pulseTrainVar array) as the analogOutVal. Lastly, "writeAnalogOutValues" writes the analogOutVals to the DACs.

These functions just homogenize a bunch of somewhat heterogeneous code. For example, csStateBehavior supports four true 12-bit analog outputs, two of which are native to the Teensy3.5/3.6 and two are added with MCP i2c analog out boards. The writes to these channels are slightly different.

Now, we place writeAnalogOutValues in the generic state body. So, you just have to add stimGen and setAnalogOutValues to any state you want to use analog output on.

Here is example code for a simple analog output state (State 7 in the default code):

```
// ****************************************
// State 7: Single Pulse Train Trial State
// ****************************************
else if (knownValues[0] == 7) {
  if (headerStates[7] == 0) {
    genericHeader(7);
    visStim(0);
    blockStateChange = 0;
  }
  stimGen(pulseTrainVars);
  setAnalogOutValues(analogOutVals,pulseTrainVars);
  genericStateBody();
}
```

You can set key parameters that determine the waveforms that go over the analog out lines over serial. On every interrupt the function "*setPulseTrainVars*" is called and it will register any changes commanded over serial. The actual processing of a variable change takes 18 microseconds, but the latency of variable changes will be determined by how fast the serial command gets to the Teensy. When using the csBehavior.py python program, the latency is typically 1-10 ms. This is more than adequate to drive variable stimulus trials. However, if you need more real-time performance I suggest programming logic in your state that responds to inputs and makes local pulse train variable changes. Under those conditions, you can change pulse train variables with sub-millisecond latency. Below I describe the setPulseTrainVars function and document the serial codes needed to change pulse train variables.

On every interrupt, the program looks for new variables over serial. These variables control a variety of things. The pulse train variables work a bit differently than the rest. Because there are up to four DAC channels, each variable needs to have a value and a target channel.

setPulseTrainVars(curSerVar, knownValues[curSerVar]);

The variables are as follows:
d: interpulse interval in milliseconds.
p: pulse duration in milliseconds.
v: amplitude of pulses in raw 12-bit value with 0V = 0 and 3.3V = 4095
t: stimulus type (0 for square waves; 1 for ramps; more to come).
m: max pulses/ramps to do in a state (optional: you can use this to limit the number of pulses)

The way you call each variable is to send the variable header letter above, followed by an integer value, then followed with an integer channel (1-4), and lastly the character '>' send as one line.

As an example, to make the pulse duration of channel 2 80 ms, we send:
'p802>'

The program knows the ones position is the channel and the rest is the value to use. The intention behind the design is that if you want to do some standard experiment where you drive an analog device like and LED etc. on a trial-by-trial basis, you can make a stimulus sate (csStateBehavior's default state 7 is an example of this) and toggle between a wait state (such as state 1) and the stimulus state. In this scenario you can set the current trial's stimulus variables in state 1 while you wait for some event to occur, like a baseline period to elapse, and then transition to state 7, where your stimulus will start. Using the python program csBehavior.py you could determine a trial's stimulus amplitude based on randomizing a range, or some other criteria, then in state 1's header you could push the variables and then once some baseline condition is met python can tell Teensy to move to state 7 and stimulate.

Using the "max pulses" variable you can limit the number of pulses delivered in a manner that is independent of time. For example, if you wanted to deliver 10 pulses of an LED, on channel 1, at 10 Hz, but have a two minute post-stim period, you could pass m101> and d1001>, then in your stimulus state you can track channel 1's max pulse counter, which is in the array *pulseTrainVars* which is a 4x9 array that has all the DAC values. *pulseTrainVars[0:3][8]* is the max pulse tracker. It decrements every time a pulse or is completed, until it gets to zero. So, you could have the following logic in your state:

```
If (pulseTrainVars[1][8]==0){
 pulseTrainVars[1][5] = 0;
}
```

pulseTrainVars[1][5] happens to be channel 1's amplitude.

It may now be helpful to document pulseTrainVars organization:

At compile time it is initialized in the following way:

```
uint32_t pulseTrainVars[][9] =
{ {1, 1, knownValues[9], knownValues[10], 0, knownValues[11], knownValues[12], 0, 0},
  {1, 1, knownValues[9], knownValues[10], 0, knownValues[11], knownValues[12], 0, 0},
  {1, 1, knownValues[9], knownValues[10], 0, knownValues[11], knownValues[12], 0, 0},
  {1, 1, knownValues[9], knownValues[10], 0, knownValues[11], knownValues[12], 0, 0}
};
```

Ugh :(

Not too worry. That code is specific to the way things are handled internally. What is important is position along a row, each row is a DAC channel. Arrays are zero indexed.

0: pulsing or waiting: this is a state variable that is used by the pulse generator to keep track of whether the channel is pulsing or baselining.

1: sample counter. This is used to keep time in the train.

2: interpulse interval duration in ms, this is the variable you can set with ‘d’ over serial. You can set it locally by writing to pulseTrainVars[chan][2]

3: pulse duration

4: baseline amplitude (defaults to 0).

5: pulse amplitude, encoded in raw 12-bit format Teensy 3.6 is 3.3V so 0 is 0V and 4095 is 3.3V and all in between.

6: stim type, 0 is square waves and 1 is ramps

7: write value, this is the calculated valued based on waveform parameters to write on that sample.

8: max pulse counter, this is a counter that decrements to zero, and can be used for control purposes.