A microcontroller 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.

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."

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. 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, 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. 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) **lickSensorAValue**: 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 does not evaluate licks in real-time, and does not need to.

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.

j-m) **genAnalogInput0-3**: these are the values of 4 different analog inputs chosen by the user.

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();
}
```

> setPulseTrainVars(curSerVar, knownValues[curSerVar]);
What this does is translates pulse train related variables that serial calls can change.