

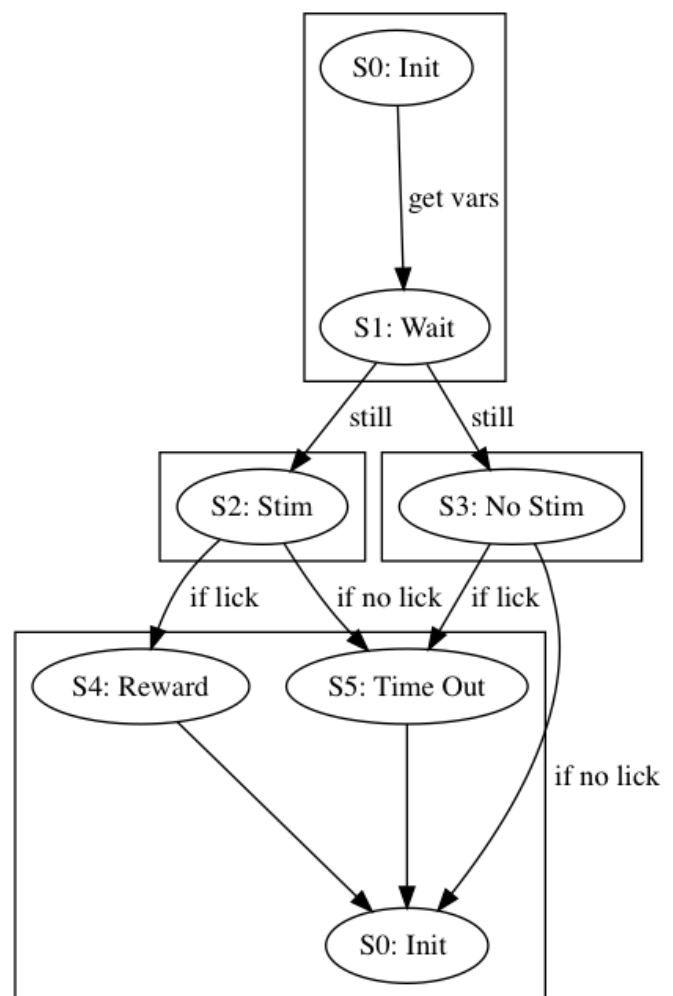
0) Introduction

csVisual is a group of tools used to implement behavioral tasks and experiments that involve the delivery of timed visual stimuli. The backbone is a program that runs on a Teensy micro-controller (PJRC.com) and a Python 3 program. csVisual is 'state-based' and the specific states active at any moment are active across all hardware and software components. The states follow a basic generic structure that allow for relatively rapid configuration and customization in order to implement different types of tasks and experiments. In order to introduce you to the basics, we will work through a very stripped down version of the Teensy script and build it up in order to implement a visual detection task.

1) Example: Implementing a Detection Task

The state diagram for our detection task is to the right. Here, we start with a boot/init state we call S0. S0 is the state we expect to find the Teensy in every time we connect to it with any other program. In S0, the Teensy will not track time or stream data to its recipient. Also, S0 is a convenient place to poll for new variables that we will use through a trial. Once we establish a connection and change variables etc., we move on to the rest of the trial. We want our subjects to start each trial in as close to the same "behavioral state" as possible. What I mean by behavioral state is that the animal is not moving around appreciably, attempting to lick for a reward, and maybe less obvious things like the eyes aren't moving around too much. In the state diagram to the right, I denote these behavioral state conditions being met as "still," just to be short about it.

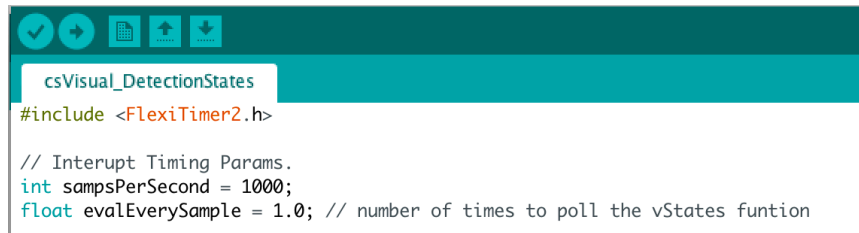
Once the animal is still, either a stimulus trial, or a catch-trial without a stimulus is offered after some variable period of time. If we selected a stimulus trial, we move the subject to S2, where a visual stimulus will be triggered and we will monitor licking at a spout. If the animal licks at the right time, in a manner consistent with its training indicating it perceived the stimulus, then it will be rewarded in S4 with water at the spout, otherwise it will move on to the next trial. If we selected a catch-trial, we move the subject to S3, where no stimulus will be presented. In S3 we will also monitor licking, if the animal licks as if to report perception we send the animal to a punishing time-out state (S5).



What I hope is clear is that in each state we more or less do the same thing. We need to keep track of time (true about any experiment), we also need to poll external variables and make decisions about when to move to a new state, or when to stay put. In addition, we need variables to be polled to tell us what to do with specific things in specific states. For example, in S1 we need to make sure the lick rate of the subject is at 0 for some specified period of time, as well as its motion is at 0 etc. In order to

maximize training, we may even want the tolerance for errant licking etc. to be wide early in training and converge to a more conservative cut-off as the subject progresses.

We start with the Teensy program and flesh out the basic core state. For reference, I will be describing a script called “csVisual_Detection” in the GitHub repository. I expect you are using a Teensy 3.6 controller, but there is critical reason for this. Frankly any microcontroller will work. What I like about Teensy boards are their size, power, and number of hardware serial lines. Moreover, Paul at PJRC maintains robust libraries that are very useful. Unlike a typical Arduino/Teensy program I do not use the main loop to run our task. Instead, we will use an interrupt timer to time calls to a task function that will behave like our main loop would. We call this by including “FlexiTimer2” (https://www.pjrc.com/teensy/td_libs_MsTimer2.html) in our program and then scheduling the timer in the setup block:



```
csVisual_DetectionStates
#include <FlexiTimer2.h>

// Interrupt Timing Params.
int sampsPerSecond = 1000;
float evalEverySample = 1.0; // number of times to poll the vStates funtion
```

After the include statement, I specify two variables necessary for flexiTimer. ‘sampsPerSecond’ sets the sample rate and ‘evalEverySample’ sets how often the interrupt will poll the function. We want it to poll every tick of the interrupt and that will occur once every thousandth of a second.

We then start the timer in the setup block (see highlighted lines):

```
void setup() {
  pinMode(syncPin,OUTPUT);
  digitalWrite(syncPin,LOW);
  Serial.begin(115200);
  delay(10000);
  FlexiTimer2::set(1, evalEverySample / sampsPerSecond, vStates);
  FlexiTimer2::start();
}
```

The reason I mention this is because you may not be familiar with this interrupt style of programming. The reason for it, is to ensure we have a reliable clock. We are going to treat this Teensy as the ‘master’ of our data-acquisition (DAQ) chain.

Note our loop block is blank, but our ‘vStates’ block is lengthy (and there are many other functions):

```
void loop() {
}

void vStates() {
  int rVar = flagReceive(knownHeaders, knownValues);
  tState = knownValues[0];

  // *****
  // State 0: Boot/Init State
  // *****
  if (tState == 0) {
    if (headerStates[0] == 0) {
      genericHeader(0);
      loopCount=0;
    }
  }

  // Some things we do for all non-boot states before the state code:
```

With that out of the way there are two main ‘types’ of states. One, is the “boot state,” which we will call State 0/S0. All other states are “Non-Boot States.”

Let’s look at S0:

```
void vStates() {  
  
    // We always first look for variable changes.  
    // We always set tState to the serial variable entry it corresponds to.  
    int rVar = flagReceive(knownHeaders, knownValues);  
    tState = knownValues[0];  
  
    // *****  
    // State 0: Boot/Init State  
    // *****  
    if (tState == 0) {  
        if (headerStates[0] == 0) {  
            genericHeader(0);  
            loopCount=0;  
        }  
    }  
}
```

On every iteration of vStates, we execute two lines, even before we execute any S0-specific code. We look for any variables that have been sent from the Python program (more on that later) using ‘flagReceive,’ which will be explained later. tState is then set. The ‘knownValues[0]’ value will become clearer later, but it is part of an array that tracks variables that can be set over serial. Python controls the states and the Teensy is to follow along. Because tState is polled every sample and before we look for which state’s code gets executed, we can change state every sample.

We initialize knownValues[0] to 0, so the Teensy will start in 0, but when you change trials or quit the Python program, you will need to set the Teensy back to 0. I will explain how to do that later.

Every state has the following structure:

- > Header: Code executed once on entry. Useful for resetting counters etc.
 - * We execute a generic header, which resets a timer offset for tracking state time.
 - * In the generic header we also set the headerState for state 0 to 1 and all others to 0.
- > Body: State 0 is missing the body, but for all other states we track total time and state time, at a minimum. But, this is where we would insert code for monitoring state and sensors. Also, data is relayed back to the Python program for parsing, plotting and saving.
- > Exit: From the Teensy side, we would rarely implement this, because we want the Python program to be in control.

We add custom code in S0 header to reset ‘loopCount,’ this is used to track how many interrupts we’ve ticked through the whole trial, and ensure the number of interrupts matches up with our time stamps.

Next we will look at the more standard states:

I end here — 1/28 ... will continue later. Check the code for now :)