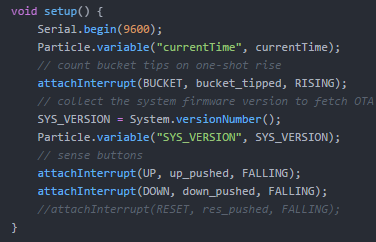
PinchValve Control System

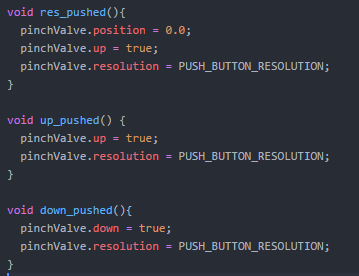
*Notes for Graham, Suyash, Aaron, and Mark*



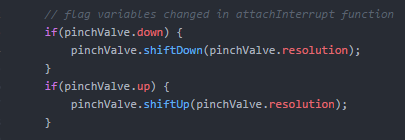
Everything begins with the setup loop in the microcontroller code. It runs this first once the cellular connection has been enabled. Notable for us are the two bottom attach interrupt functions. Essentially it will always listen for these pins to fall while in the ‘loop’ section of the code (MCUs are always in either the setup at the beginning and then otherwise always in the loop function)



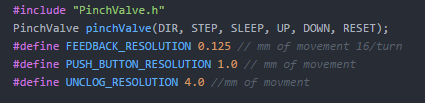
This is their syntax (the way to use them). UP and DOWN are constants set in pinmapping.h, they will execute up\_pushed function and down\_pushed function when their pin, usually held to 3.3V, falls to 0V. Reset’s attach interrupt will work similarly. These are the pins hooked up to push buttons. They only will happen when the push buttons are pressed. It’s how it’s all setup in the hardware.



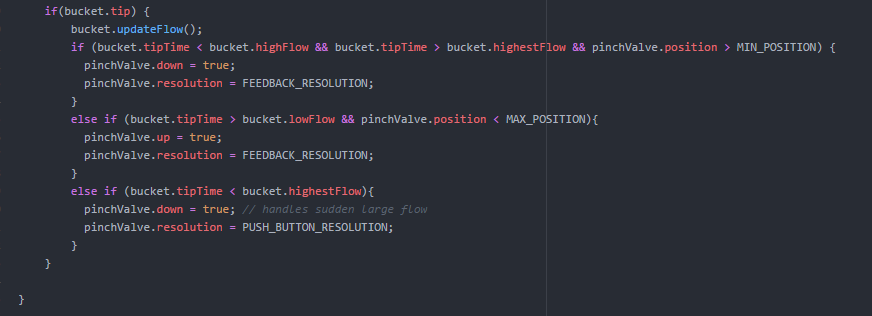
Here are those corresponding functions. There’s this ‘object’ known as pinchValve that you can think of like a box in the microcontroller code. There are functions (also known as methods) and variables you can access inside the box using the dot notation here. If up is pushed it sets to true and the resolution variable is set to push button resolution which is a constant. I can show how that is set later. These functions (down\_pushed, etc.) should never be activated long (a general rule of thumb) so I set the variables to do something that will cause a change when the MCU goes back into the ‘loop’



So here it is. We are back in the loop and if ever the pinchValve.down is true or the pinchValve.up is true (up and down are the variables inside this ‘object’ (we are calling a box)) the function inside the box will happen at the resolution that was set in the previous function! This is very dynamic because a whole range of responses could cause this such as a bucket tip or a push button pressed and it will cause the valve to move up and down by whatever is set as the resolution.

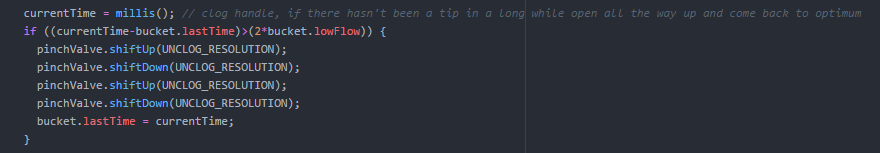


Here is the creation of the object (box analogy) pinchValve and the constants that were discussed earlier. In C++ #define is one way to set a variable. Right now we have mm of movement being our constants based on initial testing. I’m keeping it as mm of movement vs. other possible metrics because that will be constant between motors, the number of steps for the stepper motor will be variable depending on microstepping, so that is not a good metric to tell the motor to move by.

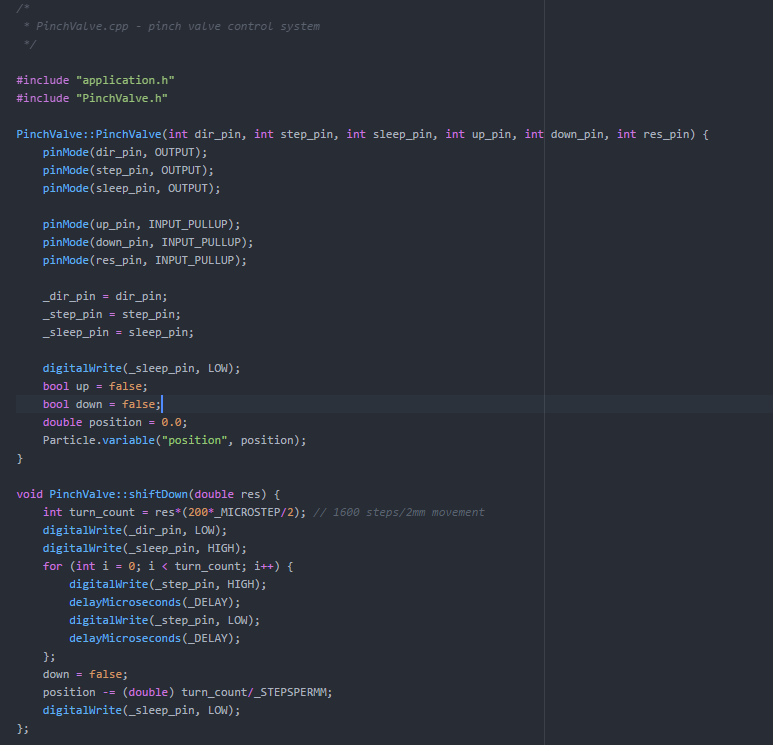


pinchValve isn’t the only box. An old structure of prior development is the tip variable. Like the reset, up, down pushed functions there is an Attach Interrupt for bucket. If you take a look up top in the setup function you’ll see it. It’s set to true to initiate the following action. Bucket has an updateFlow function that is then activated this calculates the differential of time between the current tip and the last tip. This is compared to desired values. More on this later.

Basically, if the tip time differential is really low i.e. high flow and the valve can still go lower (position is constantly monitoring) The down variable will be set to true and the resolution set to our 1/16th of a turn idea. (for Graham) This will close it down a little bit. If the opposite extreme happens it will open up. If the tip time is super high, potentially caused by a clog being released the resolution is set to the push button resolution and it will move down half a turn so it is very responsive to these sudden high flow events. Bounds you can see are set to make sure the system doesn’t move too high (reducing sensitivity to sudden declog events) also preventing the motor from having to work to hard by continuously trying to close a system when it’s already fully closed (all done with the position variable and the bounds, easily adjustable)



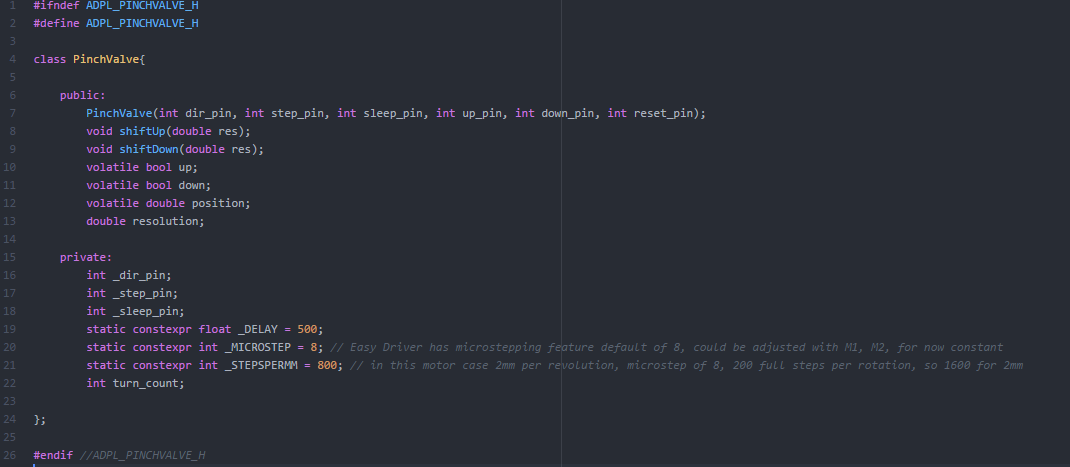
If a large amount of time has passed, maybe twice the lowest flow bound there is likely a clog in the system. The motor should fully open and fully close two times. This was an idea from Graham to help try to clear the clog. It will make sure to only do this once using the bucket.lastTime variable.



PinchValve is a class. It has two functions, shiftDown and shiftUp. Verbose naming is used throughout. The pins are initiated at the beginning as outputs. This means digital pulses will be sent from these pins. For anyone who is unfamiliar the beginning function is the constructor. This initiated when the object is created. The pins are defined as output pins and the other three which are all the attach interrupt pins, (up, down, and reset) are set as input\_pullup in agreement with Particle documentation. There is a concept of private and public variables seen in just a sec. Most of these variables are public. The ones with a ‘\_’ leading are the private variables to the class. Up and down, components of this object, are set to false initially to make sure the motor doesn’t start trying to move up and down constantly. False means resting, true means move.

As seen earlier shiftDown and Up will occur when the up/down variables are set to true. The chip on the stepper driver (the EasyDriver board) responds to a series of digital pulses on its step pin. A for loop causes these pulses to be sent in a series length corresponding to how far the valve needs to move.

Sleep functionality makes sure the driver doesn’t overwork itself and only works when it needs to. It’s usually set low and only set high when it needs to be turned on. **Motors and drivers can really overheat. It’s essential these are plugged in correctly and the firmware is coded properly. Pin outputs must stay constant whether in Kenya or the Phillipines to make sure we do not have accidents.**



The header in C++ helps us define the variables for the class and the functions that can be used. The public functions can be accessed where the object is created. So in our case this is in the ADPL\_electron.ino file for the MCU. the position, up, and down are necessary in the if statements to ensure the motor works when it’s needed and the position does not exceed bounds. Several private variables are created. A few are static. \_Delay is in microseconds, changing this will speed up the motor. \_Microstep is adjusted so that if we ever adjusted the default 8 microsteps for the EasyDriver this variable could be changed (alternatively it could become public and an input to the function if we want to hook up those lines to M1/M2 or switch to a new board with an SPI interface)