PinchValve Control System Final Report

By Daniel Hull

With Notes for Graham, Dr. Forbis-Stokes, Suyash and Dr. Palmeri and additional relevant personnel

Final Deliverables

- **Robust pinch valve software control system**
    - Attach Interrupt Structure for resetting system, moving up, and moving down using push-buttons
    - PinchValve class with adjustable motor speed, resolution (amount in mm up or down), position monitoring and logical bounds, and cloud monitoring of the position variable, and easy to adjust parameters for further optimization
    - Clog clearing control development to monitor when that may occur, adjusting net position as well if system is not clearing clog effectively.
    - Bucket tip response based on differential in the previous two bucket times with four different cases
        - Intense flow much greater than 10-15 L/hr, happens during clog clearing scenario, causes rapid closing adjustment
        - Flow is within bounds, nothing happens
        - Flow in L/hr is less than bounds, small adjustment open
        - Flow in L/hr is greater than bounds but not extreme case, small adjustment closed
    - Verbose naming performed throughout and hopefully good C++ practice
- **Hardware Development and progress toward a completed PCB**
    - Investigation into whether the EasyDriver is the right board for us, notes made for optimal usage
    - Schematic best practices observed and fixed for a readable professional job
    - Selection of new voltage regulator in accordance with Greg Mimack's recommendation
    - Protection of the relay system with diodes also in accordance with Greg Mimmack's recommendation w/ additional thoughts
    - Beginning's of selection of desirable PCB components both as SMTs and Through Holes
- **Presentation for the Gates Foundation**
- **Initial Testing**
- **Educational Resources, notably this report as to provide an opportunity for everyone to understand the MCU and the system**
- **Further Development**

I. **Robust pinch valve software control system**

A. Attach Interrupt Structure for resetting system, moving up, and moving down using push-buttons

```
void setup() {
    Serial.begin(9600);
    Particle.variable("currentTime", currentTime);
    // count bucket tips on one-shot rise
    attachInterrupt(BUCKET, bucket_tipped, RISING);
    // collect the system firmware version to fetch
    SYS_VERSION = System.versionNumber();
    Particle.variable("SYS_VERSION", SYS_VERSION);
    // sense buttons
    attachInterrupt(UP, up_pushed, FALLING);
    attachInterrupt(DOWN, down_pushed, FALLING);
    attachInterrupt(RESET, res_pushed, FALLING);
}
```

Setup Loop All on pinchValve_enhancement

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)

```
attachInterrupt(pin, function, mode);
```

Particle Syntax for attachInterrupt

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.

```
void res_pushed(){
  pinchValve.position = 0.0;
  pinchValve.up = true;
  pinchValve.resolution = PUSH_BUTTON_RESOLUTION;
  bucket.lastTime = millis();
}

void up_pushed() {
  pinchValve.up = true;
  pinchValve.resolution = PUSH_BUTTON_RESOLUTION;
}

void down_pushed(){
  pinchValve.down = true;
  pinchValve.resolution = PUSH_BUTTON_RESOLUTION;
}
```

attachInterrupt function implementation

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. This is relevant in the loop as we will see in just a second. 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'. One last note, when reset is pushed the bucket.lastTime resets so the next tip time adjustment will be based around the when the reset was pushed.

The idea for the technician in the field is he/she could move the motor to the fully closed state and press the reset and it'd move to a state that will allow flow and allow the tip time control system to do its work. Given the lack of experience of many of the technicians in the field of these low-income nations, this should be a pretty simple reset.

```
// flag variables changed in attachInterrupt funct
if(pinchValve.down) {
    pinchValve.shiftDown(pinchValve.resolution);
}
if(pinchValve.up) {
    pinchValve.shiftUp(pinchValve.resolution);
}
```

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 (mm).

B. *PinchValve class with adjustable motor speed, resolution (amount in mm up or down), position monitoring and logical bounds, and cloud monitoring of the position variable, and easy to adjust parameters for further optimization*

```
#include "PinchValve.h"
PinchValve pinchValve(DIR, STEP, SLEEP, UP, DOWN, RESET);
#define FEEDBACK_RESOLUTION 0.125 // mm of movement 16/turn
#define PUSH_BUTTON_RESOLUTION 1.0 // mm of movement
#define UNCLOG_RESOLUTION 4.0 //mm of movment
#define MAX_POSITION 5.0 // in mm
#define MIN_POSITION 0.0 // in mm
```

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. 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.

Feedback resolution, push button resolution, and unclog resolution were all variables set in correspondence with Graham who did a lot of initial testing of how flow varies as a function of of these movements about where the system is reset. The methods will translate these mm of movement into the number of turns it needs to do to get there.

```
#ifndef ADPL_PINCHVALVE_H
#define ADPL_PINCHVALVE_H

class PinchValve{

    public:
        PinchValve(int dir_pin, int step_pin, int sleep_pin, int up_pin, int down_pin, int reset_pin);
        void shiftUp(double res);
        void shiftDown(double res);
        volatile bool up;
        volatile bool down;
        volatile double position;
        double resolution;
        int clogCounting;
    private:
        int _dir_pin;
        int _step_pin;
        int _sleep_pin;
        static constexpr float _DELAY = 500;
        int _STEPSPERMM; // in this motor case 2mm per revolution, microstep of 8, 200 full steps per ro

        static constexpr int _MICROSTEP = 8; // Easy Driver has microstepping feature default of 8, coul
        static constexpr int _MMPERTURN = 2; // constant for present motor
        static constexpr int _FULLSTEPSPERTURN = 200; // constant for present motor
        int turn_count;

};

#endif //ADPL_PINCHVALVE_H
```

The header in C++ helps us define the variables for the class and the functions that can be used. The object can be accessed where the object is created. So in our case this is in the ADPL_electron.ino file for the MCU (this was defined in the figure above this). There are two public functions these are shiftUp and shiftDown. The position, up, and down are necessary in the if statements to ensure the motor works when it's called. A true variable activates it, a false closes it. The position variable monitors the position. The resolution is set externally in the .ino file to be passed into the function which tells the motor how far to turn. The clog counter will be explained more in a minute.

Several private variables are created. The pins are only accessed in the class so they don't need to be public. '_' leads all private variables. A few are static. _Delay is in microseconds, changing this will speed up the motor. _Microstep, _MMPERTURN, and _FULLSTEPPERTURN are all used to calculate the private variable _STEPSPERMM. This all is structured to understand where the number of steps per mm comes from for whoever comes after me. This could be adjusted to a new structure if we want to change the system's microstepping (not sure why this would be useful) or if we choose a new driver, or a new motor. It's important to remember the three are constants under the current motor choice and driver selection and they would need to be changed if the selections changed.

```cpp
/*
 * PinchValve.cpp - pinch valve control system
 */

#include "application.h"
#include "PinchValve.h"

PinchValve::PinchValve(int dir_pin, int step_pin, int sleep_pin, int up_pin, int down_pin, int res_pin) {
    pinMode(dir_pin, OUTPUT);
    pinMode(step_pin, OUTPUT);
    pinMode(sleep_pin, OUTPUT);

    pinMode(up_pin, INPUT_PULLUP);
    pinMode(down_pin, INPUT_PULLUP);
    pinMode(res_pin, INPUT_PULLUP);

    _dir_pin = dir_pin;
    _step_pin = step_pin;
    _sleep_pin = sleep_pin;

    _STEPSPERMM = (_FULLSTEPSPERTURN/_MMPERTURN)*_MICROSTEP;

    digitalWrite(_sleep_pin, LOW);
    bool up = false;
    bool down = false;
    double position = 0.0;
    double resolution = 0.0;
    Particle.variable("position", position);
    int clog_counter = 0;
}
```

Constructor

PinchValve has a .cpp file in pair with its header to specify the functionality of the class more clearly. 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. 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. Again, false means resting, true means move. The steps per mm calculation is made here to show where this variable comes from. The other three pins, dir, step, and sleep are set private to this class and initialized here. The position starts at 0 when the system totally resets. **This should be fixed in future iterations when the watchdog becomes available. Monitoring of this position should be available over the particle cloud.** For one not familiar with the hardware, the following pins do the following

_dir_pin - sets the direction (set in pinmapping.h) (HIGH means move a direction, LOW the other)
_step_pin - tells the chip to initiate steps if a cycle of LOW-HIGH-LOW happens that is at least a few microseconds (see the Easy Driver Chip Selection Datasheet)
_sleep_pin - ESSENTIAL, this turns on and off the driver to only work when a step pattern needs to be initiated.

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.

```
void PinchValve::shiftDown(double res) {
    int turn_count = res*(_STEPSPERMM); // 1600 steps/2mm movement
    digitalWrite(_dir_pin, LOW);
    digitalWrite(_sleep_pin, HIGH);
    for (int i = 0; i < turn_count; i++) {
        digitalWrite(_step_pin, HIGH);
        delayMicroseconds(_DELAY);
        digitalWrite(_step_pin, LOW);
        delayMicroseconds(_DELAY);
    };
    down = false;
    position -= res;
    digitalWrite(_sleep_pin, LOW);
};

void PinchValve::shiftUp(double res) {
    int turn_count = res*(_STEPSPERMM); // 1600 steps/2mm movement
    digitalWrite(_dir_pin, HIGH);
    digitalWrite(_sleep_pin, HIGH);
    for (int i = 0; i < turn_count; i++) {
        digitalWrite(_step_pin, HIGH);
        delayMicroseconds(_DELAY);
        digitalWrite(_step_pin, LOW);
        delayMicroseconds(_DELAY);
    };
    up = false;
    position += res;
    digitalWrite(_sleep_pin, LOW);
};
```

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. For this instance, we should see an 800 steps per MM calculation so if the resolution is 1 MM we will see 800 cycles of the for loop telling the motor 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.

Up and Down are set to false to turn off the movement once a cycle of the method is complete.

C. *Clog clearing control development to monitor when that may occur, adjusting net position as well if system is not clearing clog effectively.*

```
currentTime = millis(); // clog handle, if there hasn't been a tip in a long w
if ((currentTime-bucket.lastTime)>(2*bucket.lowFlow)) {
  pinchValve.shiftUp(UNCLOG_RESOLUTION);
  pinchValve.shiftDown(UNCLOG_RESOLUTION);
  pinchValve.shiftUp(UNCLOG_RESOLUTION);
  pinchValve.shiftDown(UNCLOG_RESOLUTION);
  bucket.lastTime = currentTime;
  pinchValve.clogCounting += 1;

  if(pinchValve.clogCounting >= 2 && pinchValve.position < MAX_POSITION){
    pinchValve.shiftUp(PUSH_BUTTON_RESOLUTION);
  }

}
```

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.

It will also count up one every time this process is repeated without a tip. If two have passed without a tip occurring, we will see a movement of 1 MM. Right now this is a sizeable movement. It will not move too high.

D. Bucket tip response based on differential in the previous two bucket times with four different cases

```
if(bucket.tip) {
    pinchValve.clogCounting = 0;
    bucket.updateFlow();
    if (bucket.tipTime < bucket.highFlow && bucket.tipTime > bucket.highestFlow && pinchValve.position > MIN_POSITION) {
      pinchValve.down = true;
      pinchValve.resolution = FEEDBACK_RESOLUTION;
    }
    else if (bucket.tipTime > bucket.lowFlow && pinchValve.position < MAX_POSITION){
      pinchValve.up = true;
      pinchValve.resolution = FEEDBACK_RESOLUTION;
    }
    else if (bucket.tipTime < bucket.highestFlow){
      pinchValve.down = true; // handles sudden large flow
      pinchValve.resolution = PUSH_BUTTON_RESOLUTION;
    }
  }

}
```

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)

This completely summarizes the control system:

- Intense flow much greater than 10-15 L/hr, happens during clog clearing scenario, causes rapid closing adjustment
- Flow is within bounds, nothing happens
- Flow in L/hr is less than bounds, small adjustment open
- Flow in L/hr is greater than bounds but not extreme case, small adjustment closed

The development here of course depends on the updateFlow function that was created in the bucket class. Let's cover that here.

```
#include "Bucket.h"
#define VOLUME 250.0 //300 mL, varies by location
#define OPTIMAL_FLOW 5.0 //5.0 L/hr, varies by location
Bucket bucket(BUCKET, VOLUME, OPTIMAL_FLOW);
```

This is the creation of the bucket object in the ADPL_electron.ino code. It depends on two constants to calculate all the desired control bounds, volume of the bucket and optimal flow. Consistent bucket size would help all of us. However, this has not been consistent to date. Front end software development could be made from the web address to have this information be pulled down every

time the system is reset. This would allow customization as optimal_flow will vary area to area as a function of the number of users.

```h
#ifndef Bucket_h
#define Bucket_h

#include "application.h"

class Bucket {
    public:
        Bucket(int pin, double volume, double optimal_flow);
        void tipped();
        unsigned int tip_count;
        bool tip;
        void publish();
        void updateFlow();
        unsigned long baseFlow;
        unsigned long highFlow;
        unsigned long highestFlow;
        unsigned long lowFlow;
        unsigned long timeRead;
        unsigned long tipTime;
        unsigned long lastTime;
    private:
        static constexpr unsigned long _OPTIMALBOUND = 60000; // +/- 60 seconds
        static constexpr double _HIGHESTFLOW = 15.0;
        double _VOLUME;
        double _OPTIMAL_FLOW;
};

#endif
```

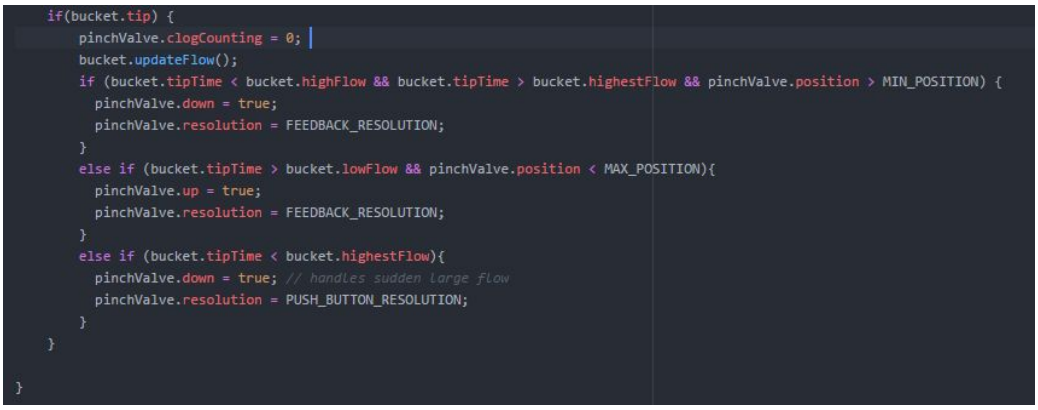
Here is the bucket header. There are three functions, tipped, publish, and updateFlow. updateFlow is the only new method. The rest of the variables will become obvious here in a second.

```cpp
/*
 * Bucket.cpp - bucket tipping counter
 */

#include "application.h"
#include "Bucket.h"

Bucket::Bucket(int pin, double volume, double optimal_flow){
    pinMode(pin, INPUT_PULLDOWN);

    baseFlow = (unsigned long) volume*3600*(1/optimal_flow);
    highFlow = baseFlow - _OPTIMALBOUND;
    lowFlow = baseFlow + _OPTIMALBOUND;
    highestFlow = (unsigned long) volume*3600*(1/_HIGHESTFLOW);

    unsigned int tip_count = 0;
    Particle.variable("bucket", (int) tip_count);
    unsigned long lastTime = 0;
}

void Bucket::tipped() {
    tip_count++;
}

void Bucket::updateFlow(){
    timeRead = millis();
    tipTime = timeRead-lastTime;
    lastTime = timeRead;
    tip = false;
}

void Bucket::publish() {
    Particle.publish(String("BUCKET"), String(tip_count));
}
```

This is the bucket class. It'd take some dimensional analysis, but you'll have to trust me that the calculation above will calculate a number of milliseconds that is the optimal base flow. The high and low flow are bounds set by a private variable that will have to be optimized in testing. Highest flow is the same calculation but with a different flow variable. It will be a much lower value in ms.

updateFlow is activated whenever a tip has occurred. This calculates a differential from the last tip which is then compared to the flow rates seen in the control logic here again.

clogCounting is set back to zero every time a tip has occurred.



```
    if(bucket.tip) {
        pinchValve.clogCounting = 0;
        bucket.updateFlow();
        if (bucket.tipTime < bucket.highFlow && bucket.tipTime > bucket.highestFlow && pinchValve.position > MIN_POSITION) {
          pinchValve.down = true;
          pinchValve.resolution = FEEDBACK_RESOLUTION;
        }
        else if (bucket.tipTime > bucket.lowFlow && pinchValve.position < MAX_POSITION){
          pinchValve.up = true;
          pinchValve.resolution = FEEDBACK_RESOLUTION;
        }
        else if (bucket.tipTime < bucket.highestFlow){
          pinchValve.down = true; // handles sudden large flow
          pinchValve.resolution = PUSH_BUTTON_RESOLUTION;
        }
    }

}
```

II. Hardware Development and progress toward a completed PCB

A. Investigation into whether the EasyDriver is the right board for us, notes made for optimal usage

The Easy Driver Board has significant advantages. It provides our desired torque at low current output. It only requires one chip, few resistors, and a single voltage regulator for the board. It does not contain any additional unnecessary components. Furthermore it provides sleep functionality and microstepping if needed but does not require those lines to be connected. This means a minimum of four lines from the microcontroller need to be made. As many boards require SPI this would require a lot more lines and coding overhead to implement. (although I do have experience should we need to)

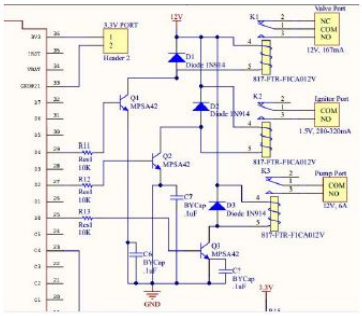
A simple potentiometer will need to be adjusted to provide our desired current before shipping off to any site. This seems to work without any adjustment though as the team did not know of its existence until recently.

B. Schematic best practices observed and fixed for a readable professional job

Drawing Rules:

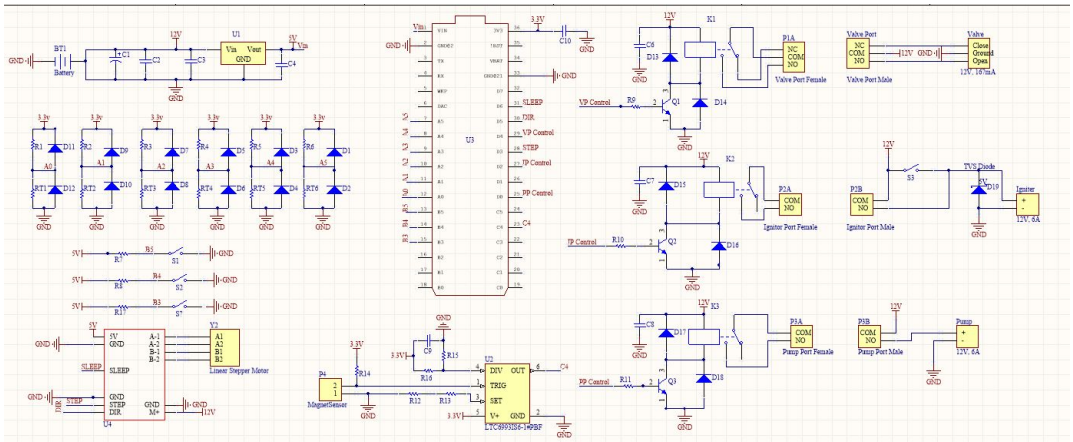
Show a logical flow of signals with input signals on the left and output signals on the right.
Higher voltages should be towards the top of the diagram with low voltages near the bottom.
Use labels for power supply and ground voltages.
Only connect power and ground labels to one component. Do not connect the same symbol to other devices.
Use a minimum of crossing wires -- none is best.

Example: From the original schematic



Greg Mimmack's suggestions

The present schematic now obeys these drawing rules.

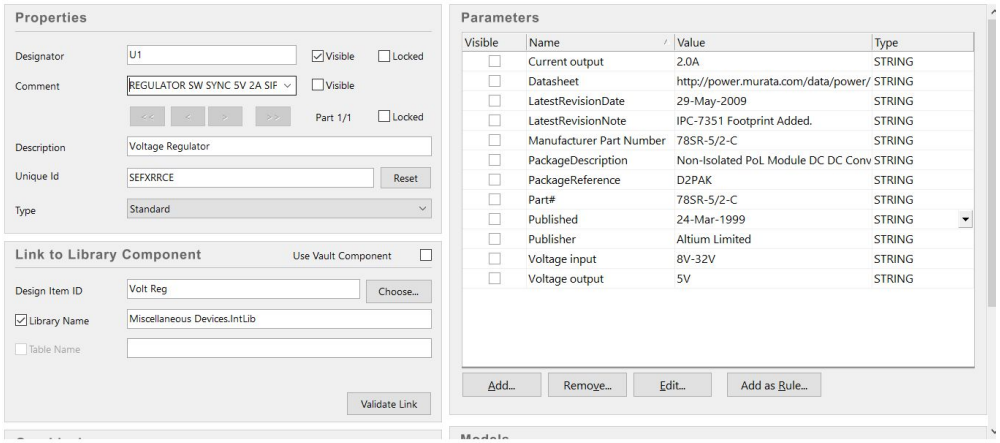


The use of Altium's nets feature prevents the cross-wiring and also allows isolation of components from the microcontroller while providing a clean representation of where the microcontroller plugs into the various lines.

Of note, all new hardware components, the stepper motor, the push button switches and the driver have been added to the schematic.

All compile errors of which there were many are now fixed.

C. *Selection of new voltage regulator in accordance with Greg Mimmack's recommendation*

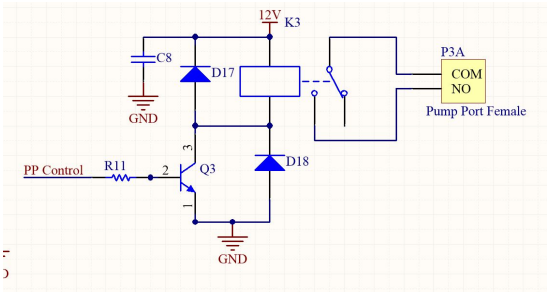


A View of the new selected Murata system with all useful fields populated (Datasheet, etc.)

According to Mr. Mimmack, our system regulator raised two problems, both confirmed. Our regulator was not supplying the recommended amount of current necessary for 2G operation of the particle system. The input voltage for the regulator does not match its lower limit as well.

Both of these issues are fixed under the new L78S09.

D. *Protection of the relay system with diodes also in accordance with Greg Mimmack's recommendation w/ additional thoughts*

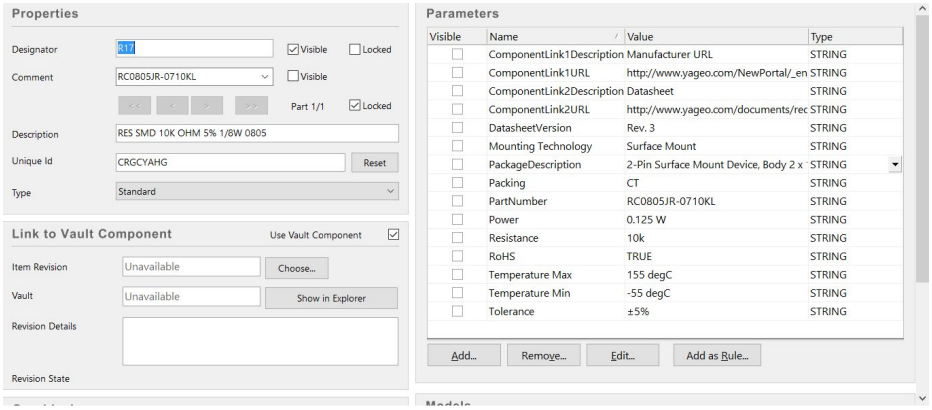


As inductors resist changes in current, they can often cause voltage spikes when the current is dropped when the transistor closes. Mr. Mimmack recommended that diodes be placed such that any voltage spike be shorted to the 12V/GND. Initially, he only put a diode represented here by D17. I

have added D18 to prevent a negative voltage spike, resulting in a shortage to ground. This was also recommended by the reference Practical Electronics.

E. Beginning's of selection of desirable PCB components both as SMTs and Through Holes

Potentially among the most valuable contributions, all components have their initial schematic information so that a PCB can begin. This will require building libraries of the components with their matching footprints. The PCB can then be constructed. I have experience with this and have built several functioning boards including one for an industry application.



All fields populated, footprint model matched as well according to IPC standards

III. Presentation for the Gates Foundation

See Dropbox GatesPresentation (for Graham to send off to Lab to edit and compile what I'd done)

IV. Initial Testing

Compiling this week

V. Educational Resources, notably this report as to provide an opportunity for everyone to understand the MCU and the system

This report!

VI. Further Development

A. Driver Combination w/ Motor

More research should go into the following to build an ideal stepper motor driver combination. I have found these recommendations from a number of engineering sources.

I Supply is >> I Load
Driver I Rat. is > 1.4X maximal I Rat. of the motor*
P Rat. is > 2*V Rat. * I Rat
V Rat. of Motor < V of Power Supply

A diagram should go into this to make sure our driver and motor combination is good. Unfortunately at present our motor has vague electrical specifications from the source we acquired it from. In the future, all motors should be bought from manufacturers with DigiKey, Mouser, or a big motor supplier. These have undergone more thorough review and must be categorized according to our desired parameters in a design process.

Additionally, a PID controller could take the place of this motor system. It was not introduced until the final weeks as a potential concept.

B. PCB Design and Manufacturing

As stated above, this should be the next direction in the hardware.