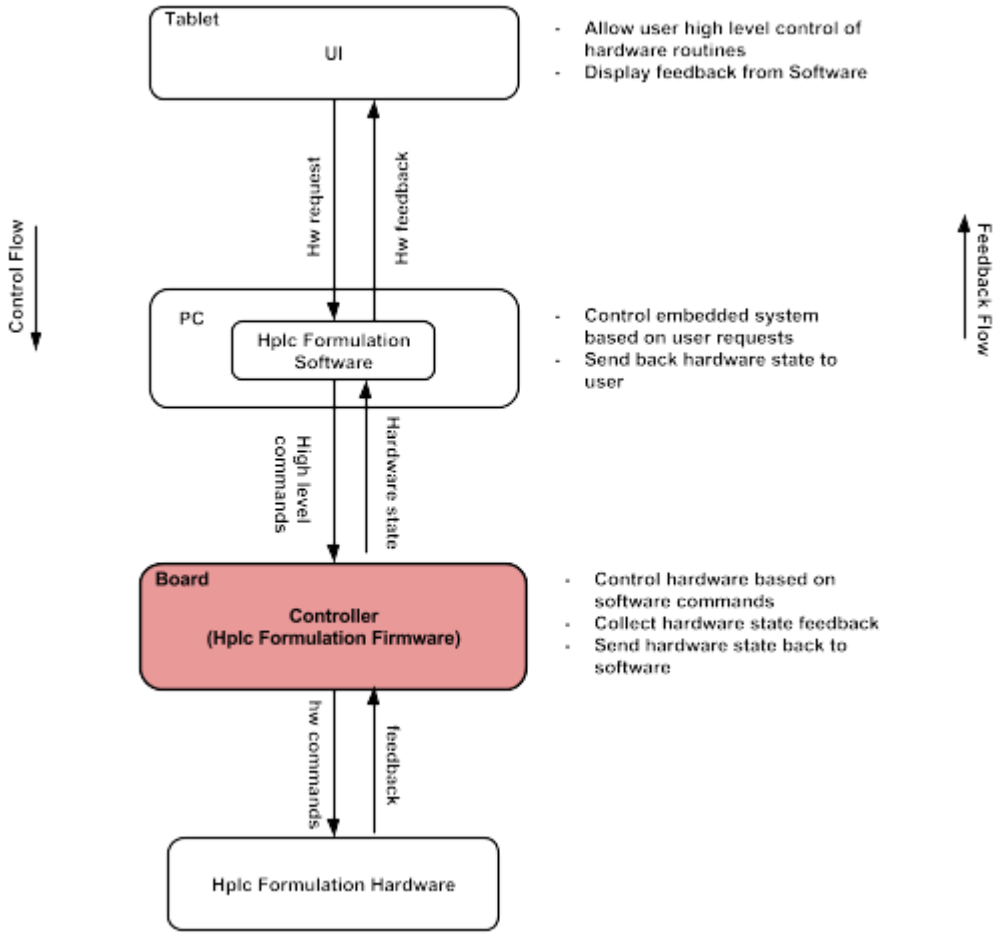
Hplc Formulation - Controller Architecture

Rev: 1.0 The architecture is based on the Hplc Formulation - Controller Requirements

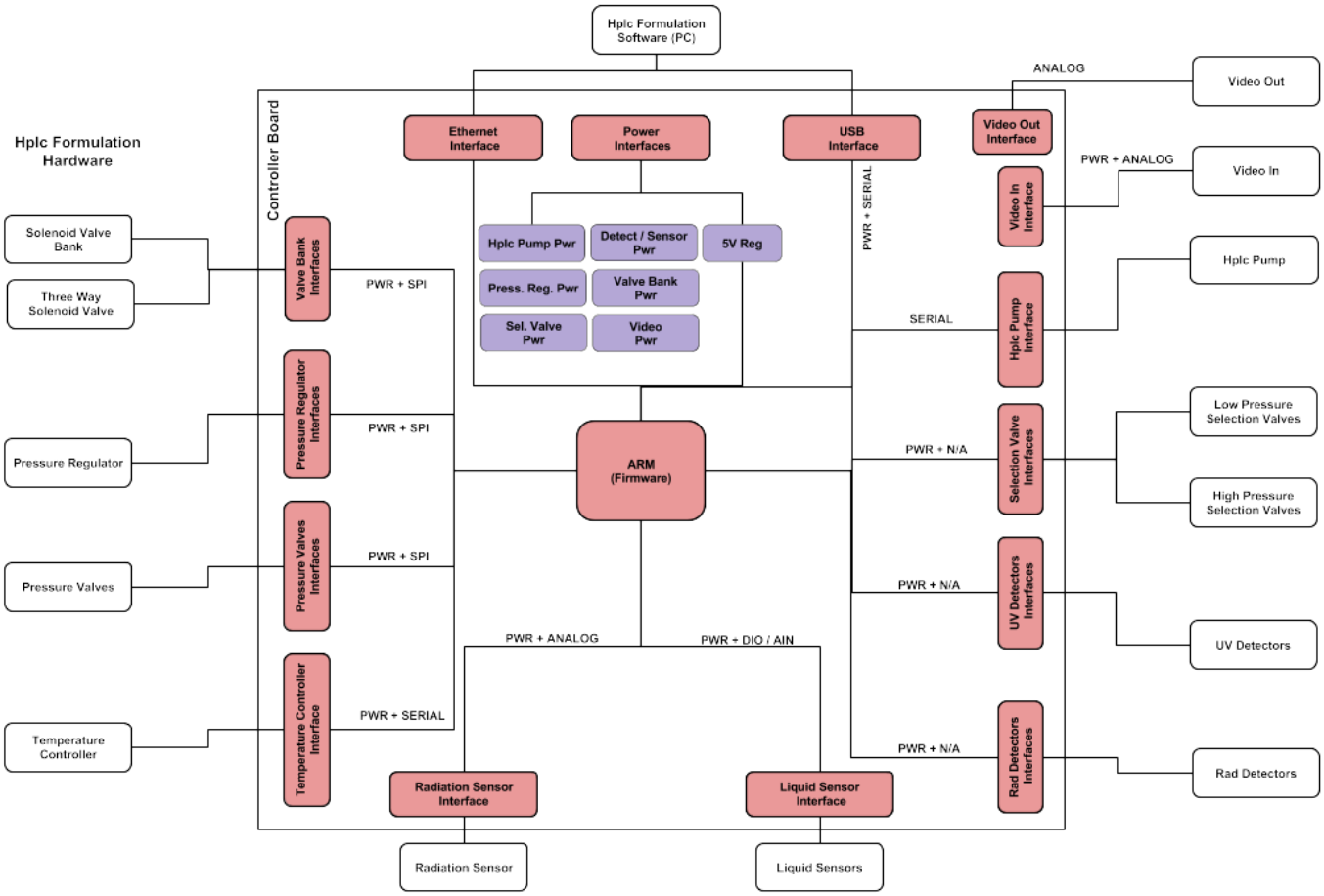
1. System Architecture

The Controller is the Hplc Formulation module embedded system responsible for all hardware functionality. Firmware running on the Controller communicates with the Hplc Formulation Hardware as well as the Hplc Formulation Software running on a separate PC.



2. Hardware Architecture

The architecture of the embedded system board (Controller) including interfaces to hardware, software, power. Each interface includes the communication and power lines to the hardware except for the Hplc Pump which only shows the communication interface.

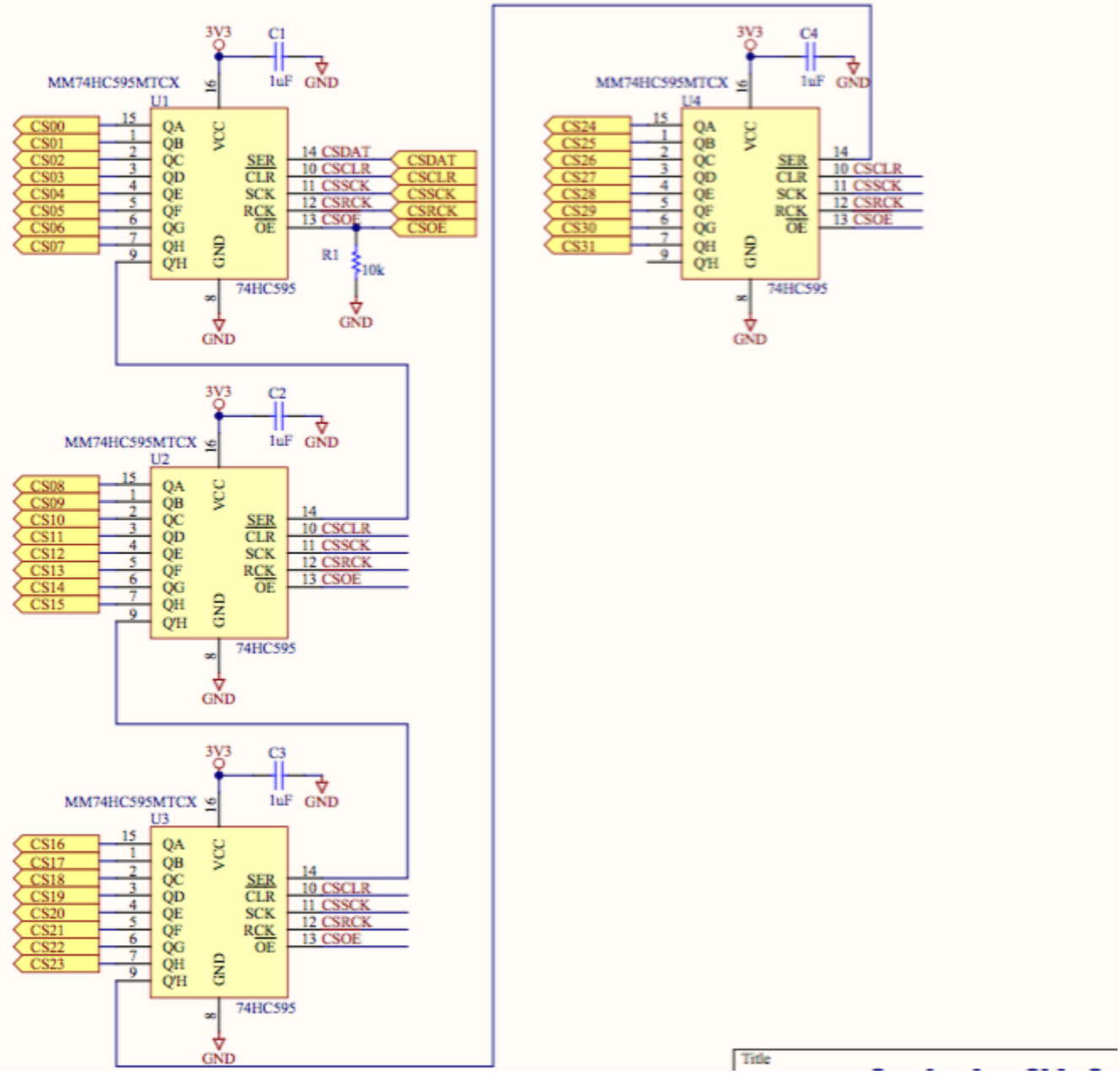


2.1. Suggested Interface Circuit Design

SPI Bus Circuit

The SPI bus will be shared by several interfaces (pressure reg, valve banks ... etc). The ARM chip controls which SPI devices are selected to communicate using the chip select (CS) line.

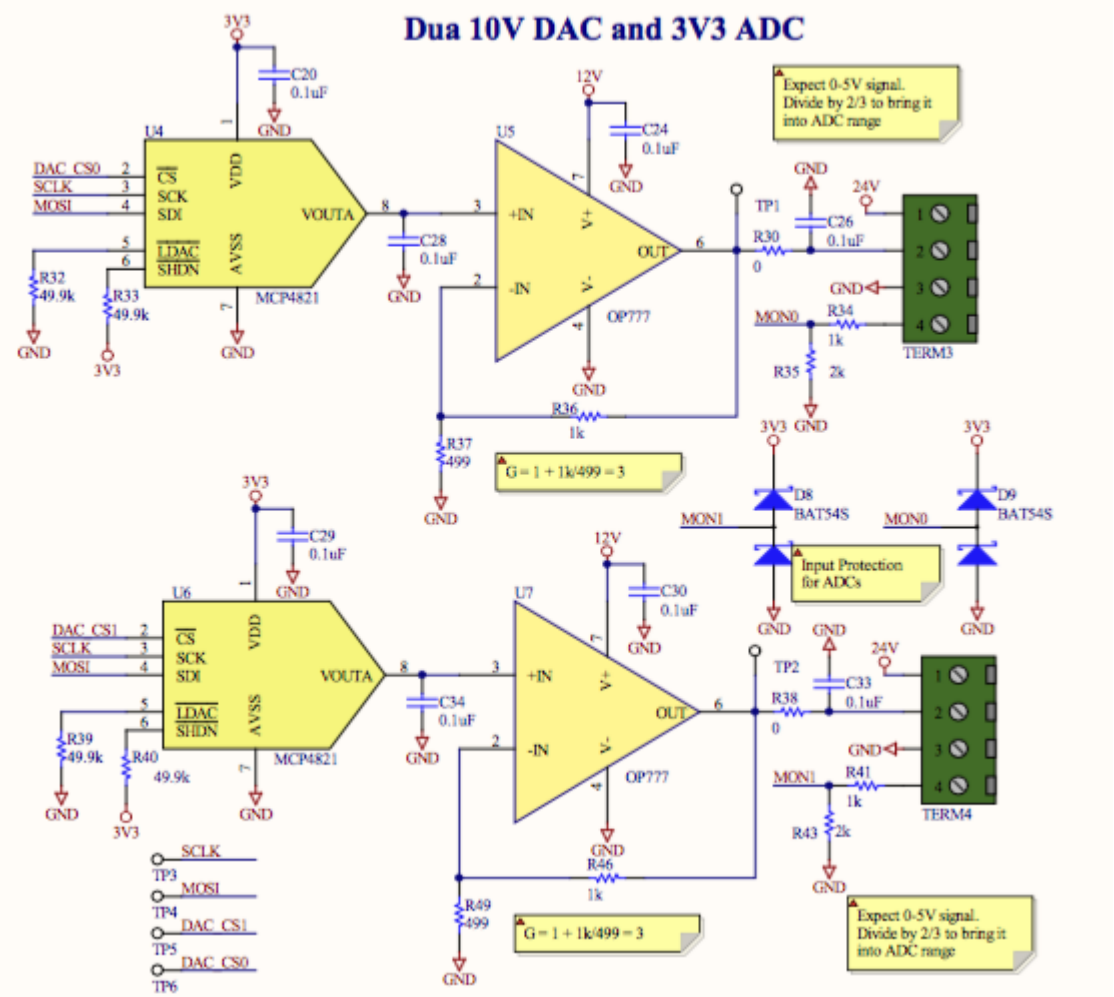
SPI Bus Circuit



Pressure Regulator Interface Circuit

Pressure regulator interface will also use an SPI controlled analog output and input circuit.

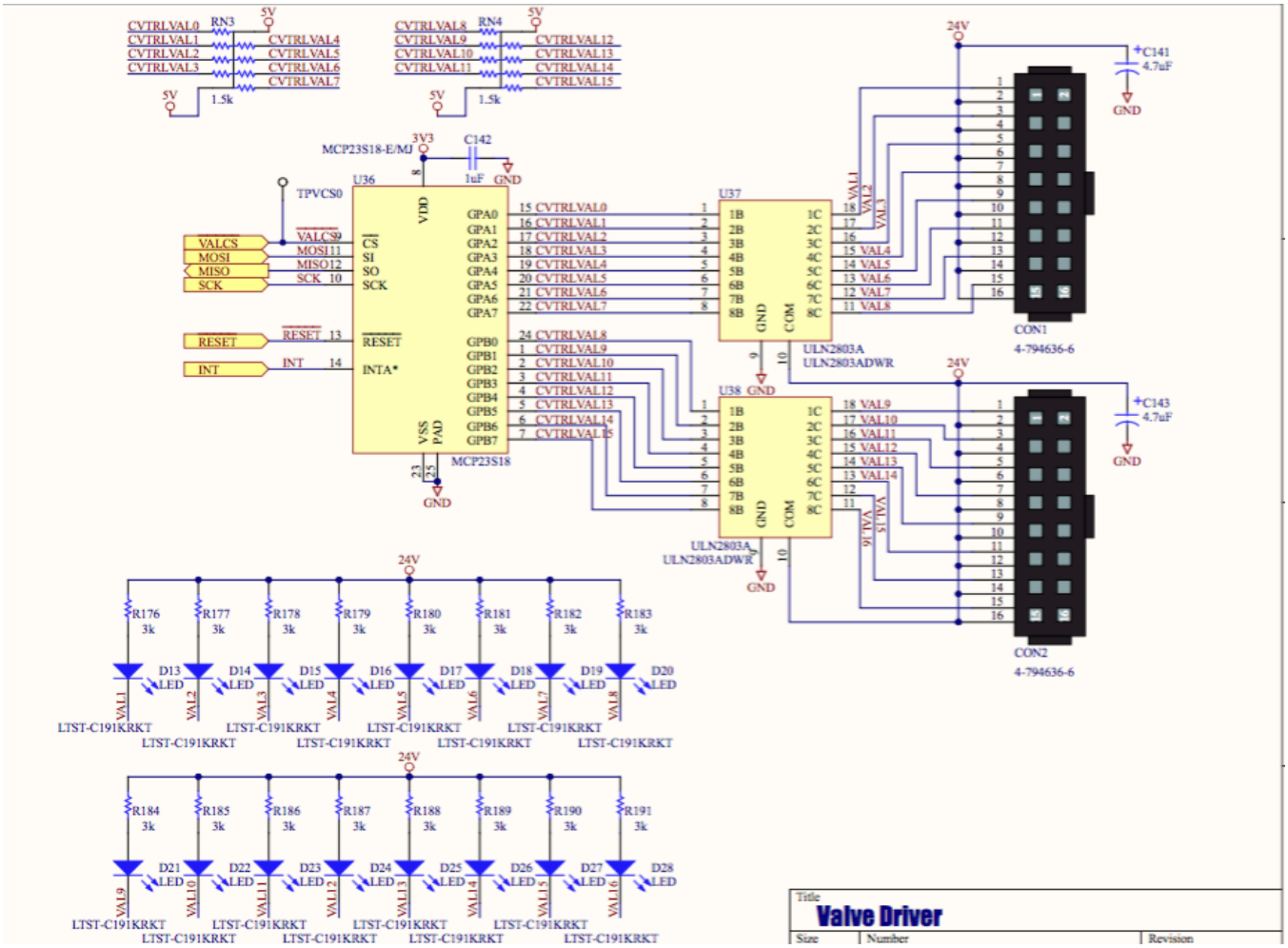
Pressure Regulator Interface Circuit



Solenoid Valve and Pressure Valve Interface Circuit

The Solenoid valve bank will be controlled by an SPI enabled shift register.

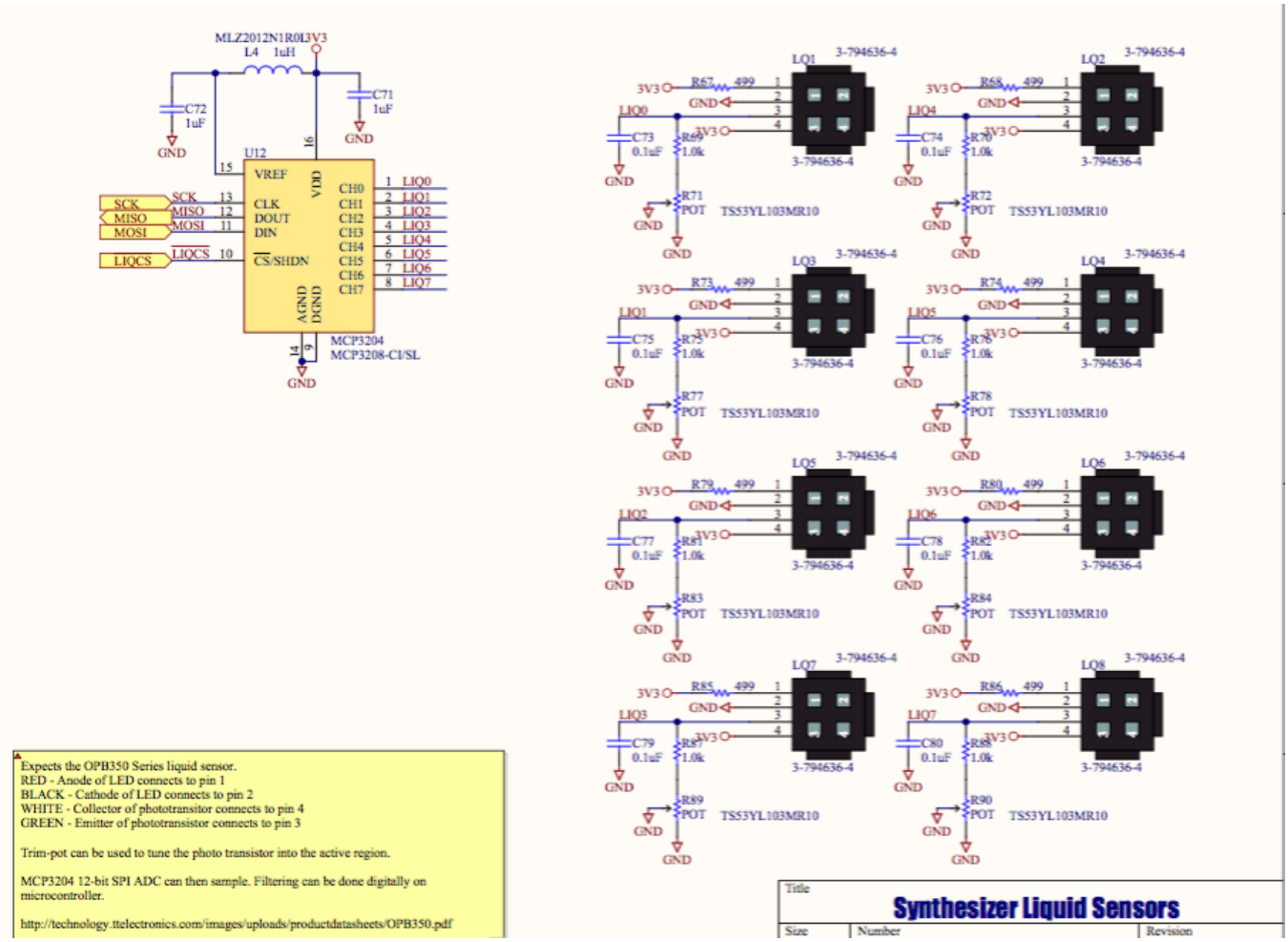
Valve Interface Circuit



Liquid Sensors Interface Circuit

The liquid sensor interface circuit includes an SPI enabled ADC responsible for reading sensor values.

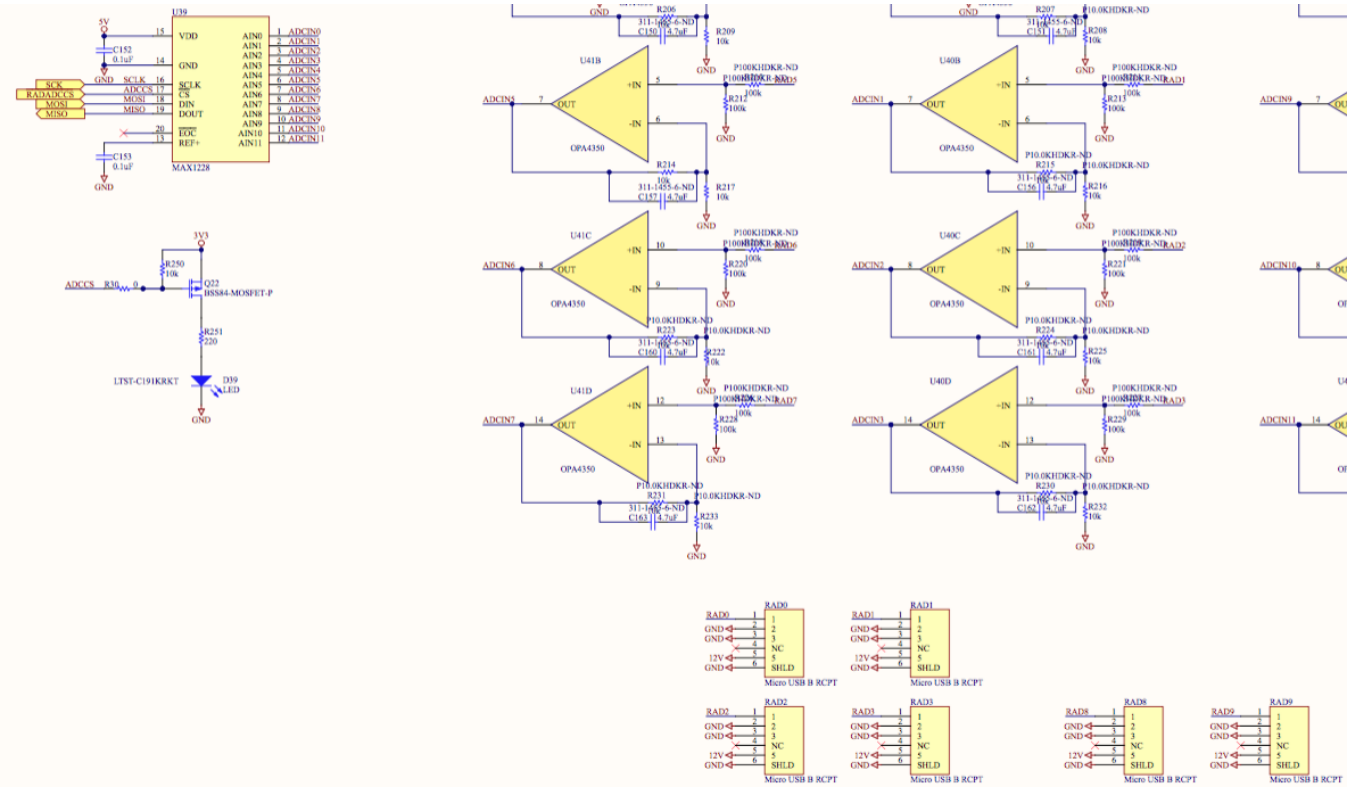
Liquid Sensor Interface Circuit



Radiation Sensors Interface Circuit

The radiation sensor interface circuit includes an SPI enabled ADC responsible for reading sensor values.

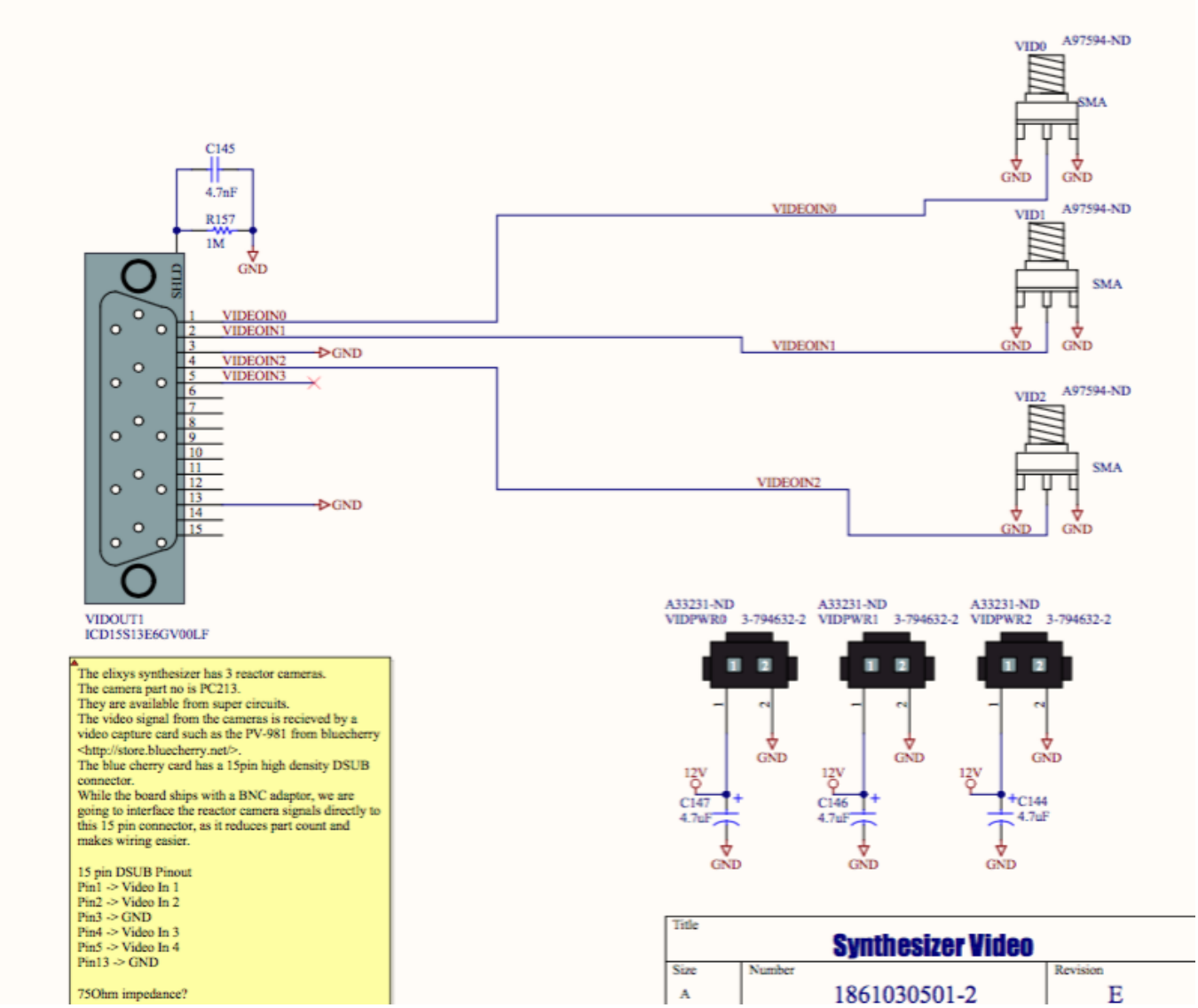
Radiation Sensor Interface Circuit



Video Input and Output Interface Circuit

The video camera input requires an SMA jack for signal routing and a separate 12V connector to power the camera. The video camera output requires a D-Sub 15 connector

Video Camera Output / Input Interface Circuit



Radiation Detector Interface Circuit

- Serial interface
- May include analog I/O

UV Detector Interface Circuit

- Serial interface
- May include analog I/O

Hplc Pump Interface Circuit

- Comm Interface: RS232 Interface to ARM chip
- Power Interface: TBD

High Pressure Selector Valve Interface Circuit

- Comm Interface: RS232 Interface to ARM chip
- Power Interface: TBD

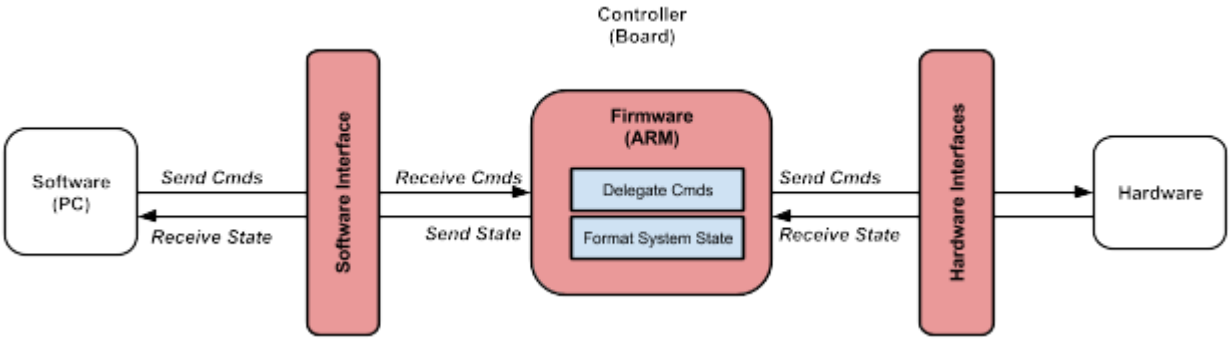
Low Pressure Selector Valve Interface Circuit

- Comm Interface: 3 solenoid ports
- Power Interface: Same as comm

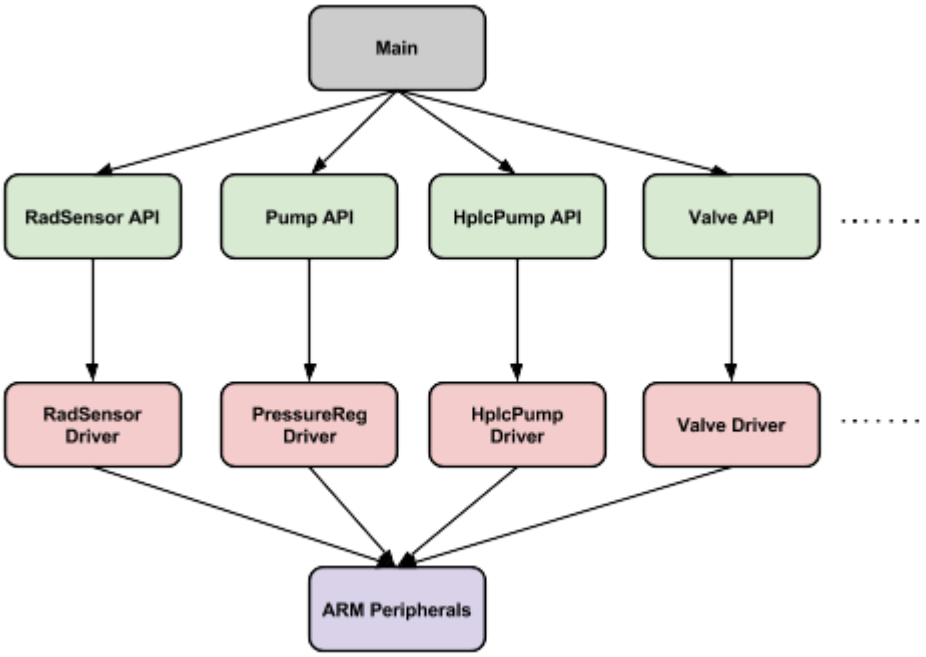
Temperature Controller Interface Circuit

- Comm Interface: Serial communication interface (UART)
- Power Interface: 5V connector, 12V connector, 30V connector, 7.5V connector

2.2. Functional Hardware Architecture



3. Firmware Architecture



Firmware Logic (Round Robin Architecture)

```
loop:
   if client command received:
       run hardware command or update hardware setpoint

   update system status
```

```
    if hardware states != to setpoints:
        send hardware commands to reach setpoint

    send system status to client
```

Expected Firmware Code (Pseudocode)

```C++

// Hardware APIs (PressureReg, Hplc Pumps, Valves, Valves Banks, Sensors, Detectors ... etc)
#include "HardwareAPI.h"

// System Status Structure - use to update with hardware state and send to PC #include "Status.h"

// if using serial to communicate with PC / software Serial usb(USBTX, USBRX); bool usbEnabled = true; //use USB as default comm

int main() { // initialize software communication interfaces usbEnabled = initSerial(); // set serial baud and return true // initialize all hardware initHardware();
```

```
// initialize system status struct
initStatus();

while true {
      if usbEnabled {
        // if cmd is available then run it before executing update state routines
and then send status
        if usb.read(&cmdPkt) {
            runCmd(&cmdPkt);
        }
        // now run routines and then send state of system back to client PC
        updateHardwareRoutines();
        // increment packet id
        updateStatusPacketID();
        usb.send(&status);
      }

      else {
       // looks like usb is not available - todo?
      }
}
```

```
}

// Functions

// Update Hardware Routines void updateHardwareRoutines() { // Read hardware setpoints, if not reached then run setpoint routines and also update status struct
pressureReg.update(status); // example: if current pressure not equal to setpoint then update pressure hplcPump.update(status); // update if setpoint not reach and then update status struct
```

valveBank.update(status); // example: if valve 0 setpoint is ON and it is currently OFF then set to ON // call other hardware updates...etc

// Sensors - just read sensor values and update status struct uvDetectors.update(status); radDetectors.update(status); liquidSensors.update(status); }

// Run Commands void runCmd(cmd_pkt_type * cmdPkt) { switch cmdPkt -> cmdID { case SET_PRESSURE_SETPOINT: float setpoint = (float) cmdPkt -> param; pressureReg.setSetpoint(setpoint); break; case SET_SEL_VALVE_POS: int pos = (int)cmdPkt -> param; selValve.setPos(pos); break; // Other hardware case routines default: // todo? break; } }

```
### Expected Firmware Hardware API Class Structure (Pseudocode)

```C++
\\Example Pressure Regulator API
#include "Status.h"
#include "PressureRegulatorDriver.h"

class PressureRegulator: PressureRegulatorDriver {
   //methods
   void setPressure(){
      float measuredPressure = sendPressure(setpoint); // use driver to set
pressure
      return measuredPressure
   }
   void setSetpoint(float pressure){
       setpoint = pressure; // this method should be called by client command
   }
   void update(Status * status){
       // send command to pressure reg to reach current setpoint and get current
pressure measured
       measured = setPressure();
       // update global status variable with new measured pressure
       status -> pressure_regulator.measured = measured;
   }
   //attributes
   float setpoint, measured;
};
```
```