Projector Controller

Developers Guide Rev. A

Author: Jesse DeGuire
Email: jesse.a.deguire+pjc@gmail.com

Date: February 22, 2013

Contents

Revision History

This section describes revisions made to this document. Changes made to the hardware, firmware, or software are described in their respective sections.

Doc Revision	Date	Changes
A	Feb. 14, 2013	Initial revision

1 Introduction

This document provides implementation details for the various pieces of the Projector Controller system. The information contain within will hopefully be useful for anyone who wishes to modify or tinker with the any part of the system or at least will satisfy an interested user's curiosity. Anyone who is only looking to get the Controller running in his or her projector should instead consult the *Projector Controller Users Guide*.

This guide is divided up into a few sections. The Hardware section describes the components used on the board itself as well as the power requirements and capabilities of the board. The Firmware section contains information regarding the program flow of the bootloader and application firmwares used by the Projector Controller. This also contains documentation for the command interfaces used by both to communicate to the host PC as well as brief descriptions of many of the source code files. Finally, the Software section describes how the PC-side software communicates to the board. All sections will additionally state what tools were used in their development so that anyone can build and modify the various pieces.

1.1 Sources and Licenses

All source files used to create the board, software, and firmwares are available in a Git repository on GitHub at `https://github.com/jdeguire/pjcontroller`. The repository also contains datasheets for the hardware components and other reference material. This document will often reference paths within the repository. For example, the path `firmware/common/uart.c` should be read as "`[locationtowhichrepositorywascloned]/firmware/common/uart.c`".

The firmware and software, the sources of which are located in the `firmware/` and `software/` directories in the aforementioned repository, are made available under the GNU Public License Version 3. A copy of the GPLv3 is provided in the file `COPYING.txt` in the repository root or online at `http://www.gnu.org/licenses/`.

This document, the *Users Guide*, their associated LaTeX source files, and the board design files are all made available under the Creative Commons Attribution-ShareAlike 3.0 Unported License. The source files for the *Developers Guide* and *Users Guide* are available in the `docs/tex/` directory. The board design files are found in `hardware/pjc-kicad/`. A copy of the Creative Commons BY-SA 3.0 license is provided in the file `CCBY-SA3.0legalcode.txt` in the repository root or online at `http://creativecommons.org/licenses/by-sa/3.0/`.

Where attribution is required, it is sufficient to include a statement of reasonable visibility in the derived work or supporting documentation mentioning that the work is derived from this project and giving this author's name. It would be appreciated if the statement also contained the URL, given above, for the version control repository.

Located in `hardware/pjc-kicad/libs/` are two files `SparkFun.lib` and `SparkFun.mod`. These were converted into their current form for use with KiCad from the original Eagle libraries provided by SparkFun Electronics under the CC BY-SA 3.0 license. The latest such libraries can be found on GitHub at `https://github.com/sparkfun/SparkFun-Eagle-Libraries`.

1.2 Open Source Hardware

As described above, all design and source code files for Projector Controller are available under licenses allowing one to "study, modify, distribute, make, and sell the design or hardware based on that design". The project was developed using freely-available tools, most of which are also open source, lowering the barrier to accessing the design. Therefore, the author believes that the Projector Controller qualifies as Open Source Hardware as given in the Open Source Hardware Definition 1.0. The defintions is availabe from the Open Source Hardware Association at `http://www.oshwa.org/definition/`.

Note that the logo itself does not have a license associated with it, but the guidelines presented at `http:`

//www.oshwa.org/faq/ recommded its use. Any derived works which do not adhere to the definition of Open Source Hardware should remove the logo from any files which contain it.

2 Hardware

The hardware was designed using KiCad EDA, an open source (GPLv2) multi-purpose EDA tool available for various flavors of GNU/Linux, Windows, and OS X. Get KiCad from the official site at http://www.kicad-pcb.org/display/KICAD/Download+Kicad. If possible, download a Snapshot build as those contain additional features over the rather old stable release. Ubuntu users should use the Adam Wolf PPA given on that page.

Datasheets for the various components used are available in the repository under `hardware/datasheets`. The datasheets are provided for reference only and belong to their respective manufacturers.

2.1 Hardware Changes

This section lists the changes that have been made to the different board revisions over time. The revision is incremented when the board layout is changed and those changes are made into manufactured boards.

HW Revision	Date	Changes
A	Feb. 14, 2013	Initial revision

2.2 Design Considerations

Becuase the Projector Controller will probably never acheive high-volume production, the main consideration was ease of assembly. When possible, the components used are of a large form factor (such as 1206 for resistors) and the footprints for surface-mount devices use long pads to make accessing them with a soldering iron easier. The board uses a socket for the 28-pin DIP microcontroller to allow easy replacement. These choices allow the board to be assembled by hand for small production runs and allow the user to more easily tinker with the board.

Though not a primary consideration, the board was also designed with an eye on power consumption. The board will be able to run certain componenets from USB power. USB provides 5V at 100mA by default, though a device can request up to 500mA. The USB interface device does not identify itself as a high-power device by default, so all 5V components will need to consume less than 100mA combined. The power consumption of the board is characterized in the Power Usage section.

2.3 Circuit Details

Following are desciptions of the different circuit pieces developed for the board. The reader of this section may want to have the schematic handy in order to follow along. The titles of the descriptions will closely match the text titles used in the schematic for the different circuits.

2.3.1 Microcontroller

The heart of the Projector Controller is an Atmel ATMega328P 8-bit microcontroller. The MCU has 32KB of flash, 4KB of which is used for the bootloader, 2KB of RAM and several useful peripherals, such as 10-bit ADCs, SPI, PWM outputs, timers, and quite a few IO pins. The MCU is a 28-pin DIP package which will insert into a socket on the board, making removal and replacement easy.

An ECS 20MHz crystal oscillator provides the CPU clock.

Three LEDs–red, yellow, and green–are provided to allow the firmware to convey diagnostic information while running. The LED outputs also control low-side drivers, which is explained in the Relay and LED Outputs description.

A jumper, pulled low, is connected to an IO pin on the microcontroller. Shorting the jumper sets the state of that IO pin high. The state of this pin is read by the bootloader when it first starts up. If the pin state is high, the bootloader will not start the application and will instead stay in the bootloader. This allows one to recover from a problematic application that fails on startup.

2.3.2 USB/UART IO

The Projector Controller communicates with the outside world by utilizing an FTDI FT232R UART to USB interface chip. The chip connects to a standard USB Mini-B connector on one side and the microcontroller's logic-level UART pins on the other, acting as a translator between the two protocols. The chip presents itself to the connected USB host as a virutal COM port to allow one to communicate with the board using a terminal program such as TeraTerm or HyperTerminal. Newer operating systems should have drivers for the device pre-installed; however, drivers are also available from http://www.ftdichip.com/FTDrivers.htm.

The circuit used is similar to the one presented in Section 6.3 of the FT232R datasheet. Because the 5V components on the board can be powered over USB (see the Main Power Input description), this device needs to comply with USB suspend mode requirements. When the USB host enters USB Suspend Mode, the attached device must power down and draw less than 2.5mA of current. To facilitate this, pin 14 of the FT232R defaults to an active low "Power Enable" pin. When entering USB suspend mode, that pin will be pulled high by the 10kΩ pullup and will "turn off" the Diodes Inc. DMP2215L P-channel MOSFET, essentially cutting power to the rest of the 5V system. When the USB host exits Suspend Mode, the FT232R will pull the line low to re-active the MOSFET. The datasheet for the MOSFET give a current rating of 2A at 70℃.

The FT232R defaults to USB low-power mode, which requires that the device in question draw at most 100mA. Given a thermal resistance of 115℃/W and on-resistance of 100mΩ, a 100mA draw would raise the MOSFET's temperature over ambient by

$$\begin{aligned}\Delta T &= P * R_{th} \\ &= (I^2 R) * R_{th} \\ &= (0.01 * 0.1) * 115 \\ \Delta T &= 0.115^{\circ} C\end{aligned}$$

Both the USB connector and the FT232R are fine-pitch parts, the latter using a 28-pin SSOP package. These two are probably the most challenging parts on the board to solder, so they will need to be placed appropriately to help make soldering them a bit easier.

2.3.3 Main Power Input

The board accepts a nominal 12V supply through a screw terminal rated up to 7A input, which is much higher than the continuous current the board is designed for (see the Power Usage and Fan Control descriptions).

To help prevent damage from occurring due to hooking up the power supply backwards, a Rohm Semiconductor RSQ035N03 N-channel MOSFET is placed across what is normally the current return path and the gate connected through a 3.3kΩ resistor to the normally-positive side. When the power is connected correctly, the gate is pulled up to 12V and the FET is fully-on. When the power is connected backwards, the gate is pulled low and the MOSFET is turned off. The MOSFET has a steady-state current rating of 3.5A and a peak rating of 14A. The device also has a max on-resistance of 62mΩ. With a maximum temperature of 150℃ and a thermal resistance of 100℃/W, at 40℃ ambient[1] the maximum power dissipated by the device is given by

[1] The inside of a projector will certainly be warmer then the outside air temperature, with the actual ambient temperature depending on where inside the projector the board is placed. The chosen 40℃ value is probably on the warmer side of normal, but provides some safety margin. This is used as the ambient temp throughout this document.

$$
\begin{aligned}
P_{max} &= \frac{T_{max} - T_{amb}}{R_{th}} \\
&= \frac{150 - 40}{100} \\
P_{max} &= 1.1W
\end{aligned}
$$

which gives a maximum current of

$$
\begin{aligned}
I &= \sqrt{\frac{P}{R_{th}}} \\
&= \sqrt{\frac{1.1}{0.062}} \\
I &\approx 4.21A
\end{aligned}
$$

meaning that the device is limited by the specified limit of 3.5A rather than by heat.

The 5V supply comes from an ST Microelectronics L78M05C linear regulator fed from the 12V supply. The device is rated to output currents up to 0.5A and features thermal overload shutdown that will cut output power in order to protect itself. The device has a maximum temperature of 150℃ and a worst-case thermal resistance of 100℃/W, assuming minimal heatsinking.

As previously stated, the 5V devices are designed to draw at most 100mA so that they can be powered via USB. A selector switch allows the user to select between USB and the 5V regulator as the 5V power source. This is useful in the case the user needs to remove the board from the projector and connect it to the PC for configuring it. The switch is rated for 6VDC at 300mA or 30VDC at 100mA. The output side of the selector switch has a Cooper Bussmann PTS120630V012 PTC fuse rated at 120mA at 23℃. Using the derating curve in the datasheet shows that the hold current drops to about 100mA at an ambient temperature of 40℃. A Diodes Inc. S1M-13-F diode, rated at 1A continuous and 1000V reverse DC voltage, is placed between the regulator and 12V input to prevent USB current from back-feeding through the regulator. If only a USB cable is connected to the board, only the 5V devices will be powered.

Given a worst-case thermal resistance of 100℃/W and nominal voltage drop of 7V, a 100mA draw would raise the device's temperature over ambient by

$$
\begin{aligned}
\Delta T &= P * R_{th} \\
&= (V * I) * R_{th} \\
&= (7 * 0.1) * 100 \\
\Delta T &= 70°C
\end{aligned}
$$

Some heatsinking can be provided by englarged pads or copper pours on the board.

2.3.4 Programming and Reset

The ATMega328P microcontroller can be reprogrammed using a standard 10-pin AVR ISP header. The pinout used should work with most, if not all, external programmers. The microcontroller is programmed through the SPI lines while the device is held in reset. The 5V pin on the header allows the board to power the programmer so that the logic signals are at the correct voltage levels. If the board is powered via USB, care must be taken to not exceed a total of 100mA draw when powering the programmer in this manner. See Power Usage for more information.

The user can reset the board using a push button connected to ground. The device's $\overline{\text{RESET}}$ pin is active low, so it is normally pulled high and pressing the button creates a connection to ground.

2.3.5 Analog Inputs

Two of the microcontroller's analog inputs are used to monitor the 12V input and ambient temperature from an Analog Devices TMP36 temperature sensor. The microcontroller's analog inputs are 10-bit single-ended ADCs.

The 12V input is divided down using a simple resistive divider in an approximate 3:1 ratio, meaning that the input into the ADC is about 1/4 the actual voltage. More specifically, to determine the actual voltage on the 12V input, multiply the ADC voltage by 4.03.

The TMP36 "F Grade" part used here has a worst-case accuracy of ±3℃. At 0℃ the device outputs 500mV and the output changes by 10mV/℃.

2.3.6 Relay and LED Outputs

Four 12V outputs are provided to control external devices. The outputs are controlled using two ON Semiconductor NUD3124DMT1G dual low-side drivers. The drivers are meant to be single-chip solutions for driving small motors and relays and so include a flyback diode and gate resistor.

The 4 connectors are 2-pin TE Connectivity Mini CT series connectors, which are 1.5mm pitch and rated for 2A maximum. The LED connectors will be grouped together and the relay connector separated a bit to help differentiate them.

Three of the driver outputs are controlled by the three LED outputs on the microcontroller (see Microcontroller description). The user can connect external LEDs to these outputs and mount them to the projector's chassis so that the firmware's diagnostic signaling is visible from outside the projector. The outputs provide 12V power, so the user will need to use an appropriate current-limiting resistor with the external LED. Alternatively, electronics stores sell LEDs packages with the proper resistor built in. The current-limiting resistors for the onboard LEDs also act as pulldowns for these outputs.

The fourth output is used as a lamp enable output and can be used to turn on, for example, an external relay coil which in turn turns the lamp on. This output has a 10k pulldown on it.

The high side of each output has a Cooper Bussmann PTS120630V012 PTC fuse rated at 120mA at 23℃. Using the derating curve in the datasheet shows that the hold current drops to about 100mA at an ambient temperature of 40℃. Given a thermal resistance of 329℃/W and worst-case on-resistance of 1.1Ω, both outputs drawing 100mA would raise the driver's temperature over ambient by

$$\begin{aligned}\Delta T &= P * R_{th} \\ &= (I^2 R) * R_{th} \\ &= (0.04 * 1.1) * 329 \\ \Delta T &\approx 14.5°C\end{aligned}$$

2.3.7 Fan Control

A single Molex 4-pin fan header is provided for connecting a standard PWM-controlled PC fan. The connector is rated at 4A max per contact. The Sense output, which acts as a tachometer signal from the fan, is not used. 12V power is controlled by an Infineon BSP762T high-side driver. The driver has a worst-case on-resistance of 200mΩ. With a maximum temperature of 150℃ and a thermal resistance of 95℃/W, at 40℃ ambient the maximum power dissipated by the device is given by

$$
\begin{aligned}
P_{max} &= \frac{T_{max} - T_{amb}}{R_{th}} \\
&= \frac{150 - 40}{95} \\
P_{max} &\approx 1.15W
\end{aligned}
$$

which is below the rated absolute maximum of 1.5W and therefore gives a maximum current of

$$
\begin{aligned}
I &= \sqrt{\frac{P}{R_{th}}} \\
&= \sqrt{\frac{1.15}{0.2}} \\
I &\approx 2.4A
\end{aligned}
$$

which matches the typical load current ($I_{L(nom)}$) provided in the datasheet. Note that the on-resistance value used will occur only when the device is operating at its max junction temperature; the on-resistance, and therefore temperature, will usually be much lower. Users will be instructed to draw 2A max.

The driver has a thermal shutdown feature to protect the device in over-current or short circuit conditions. The device will turn the output off when in thermal overload and turn it back on when it has cooled sufficiently. In the case of short circuit, the driver will repeatedly turn off and back on with a repetitive short circuit current of 7A. The driver is not designed to continuously operate in this state, so the user will need to take care to not leave the output shorted.

The PWM output signal is supplied from one of the microcontroller's 8-bit PWM outputs. According to the Intel 4-pin fan specification (available in the `/hardware/datasheets` directory of the repository), the nominal PWM output is 25kHz. The PWM signal goes through a TI SN74LVC1G07 non-inverting open-drain buffer since the fan pulls up the signal internally. The buffer has a max switching time of 3.5ns, which is far faster then required by the PWM signal. The Intel fan spec recommends a current sink capability of 8mA. The recommended max for the buffer is 32mA (the absolute max is 50mA), so a user could theoretically control 4 fans using splitters assuming they do not overload the high-side driver.

If the connected fan does not have a PWM input signal (standard PC 3-pin fan), the Projector Controller will still provide basic on/off control with the high-side driver.

The driver input and buffer input signals each have a 10kΩ pulldown.

2.3.8 Temperature Input

Temperature is read using a Vishay NTCALUG02A103G 10k (±2% at 25℃) thermistor, which is enclosed in a ring terminal for easy mounting in the projector. The ring terminal accepts a metric M3 screw. The user will normally want to place the thermistor as close to the monitored object, usually the LCD panel, as he or she can. The user may need to try a few locations to find the best one.

The thermistor input is a 2-pin JST PH Series connector, which is 2.0mm pitch and rated for 2A maximum.

The equation for getting the temperature from a thermistor is the Steinhart-Hart equation, which is

$$
T = \frac{1}{A + B\ln(\frac{R_T}{R_{25}}) + C\ln^2(\frac{R_T}{R_{25}}) + D\ln^3(\frac{R_T}{R_{25}})} - 273.15
$$

where R_T/R_{25} is the ratio between the thermistor's actual resistance and its reference resistance (10kΩ in this case). The basic equation gives the result in Kelvin, so 273.15 is subtracted to convert that to Celsius. To

get the temperature, it is necessary to determine the thermistor's actual resistance and that can be done using the voltage going into the ADC.

The ADC is a Microchip MCP3301 differential SPI ADC that returns data in 13-bit two's complement representation. The positive input to the ADC, $IN+$, is the output of a voltage divider formed by the thermistor and a 10kΩ resistor. The ADC has a separate reference voltage input, V_{REF}, and the input range is $\pm V_{REF}$. To take advantage of the full 13-bit scale, a voltage divider with two 10kΩ resistors splits the 5V input in half and the resulting 2.5V is fed into both the V_{REF} and $IN-$ inputs. If the thermistor is 25℃, it will also be at 2.5V and so the ADC reading will be 0. The ADC reading will be negative if the thermistor is colder than 25℃ and positive if warmer. To get the actual voltage going into the ADC positive in put, convert the ADC reading to voltage (remember to sign-extend the reading) and then add 2.5V to it.

Note that if the thermistor is not connected, the voltage at the ADC should be 0 and so the reading should be negative full-scale. Conversely, if the thermistor input is shorted, then the voltage should be around 5V and so the reading should be full-scale. However, the ADC is not perfect and so such conditions may be close to, but not exactly, the full-scale conditions.

The thermistor's actual resistance can now be calculated with

$$R_T = \frac{R_2 * V_{in}}{V_{ADC}} - R_2$$

which is just the standard resistor divider equation solved for $R1$ (which is R_T in this case). To get the ratio between the actual resistance and the reference resistance required by the Steinhart-Hart equation, the above equation can be divided by R_{25}. Doing that and plugging in the appropriate values for V_{in} and R_2 yields

$$\begin{aligned}
\frac{R_T}{R_{25}} &= \frac{\frac{10k*5}{V_{ADC}} - 10k}{R_{25}} \\
&= \frac{\frac{50k}{V_{ADC}} - 10k}{10k} \\
\frac{R_T}{R_{25}} &= \frac{5}{V_{ADC}} - 1
\end{aligned}$$

Notice that when the ADC reading is at negative full-scale (-4096 counts), V_{ADC} will be zero and so will cause a divide by zero error. As mentioned above, this happens when the thermistor is shorted and is an abnormal condition. To avoid the divide by zero and other problems, clamp the ADC reading input to ±4095. Firmware or software can look for the negative full-scale reading and treat that as a special case.

The cofficients for the Steinhart-Hart equation come from a Microsoft Excel spreadsheet on Vishay's website at http://www.vishay.com/resistors-non-linear/curve-computation-list/. The spreadsheet for the NTCALUG02A103G is available in the repository at `/hardware/datasheets/VishayThmCalc.xls`. The coefficients are on the RTCOEF page of the spreadsheet and are

$$\begin{aligned}
A &= 0.0033540164 \\
B &= 0.0002565236 \\
C &= 2.605970e-6 \\
D &= 6.329261e-8
\end{aligned}$$

These coefficients and the previous equation provide enough information to evaluate the Steinhart-Hart equation.

2.3.9 Projector On Input

The controller card for the projector's LCD panel also controls the LCD's backlight and will output a 5V or 12V signal to a backlight inverter to turn it on. Since the projector discards the backlight in favor of a high-power lamp, the Projector Controller has an input for that backlight signal and will use that for the similar purpose of turning on the lamp. The input connector is a 3-pin TE Connectivity Mini CT series connector, which is 1.5mm pitch and rated for 2A maximum. A 3-pin connector was used to differentiate it from the output connectors; the middle pin is not used.

The used connector pins go to the input of a Vishay VOM618A-4T opto-isolator. The opto has a minimum current transfer ratio (gain) of 160% when the current through the LED is 1mA. The output is a collector-emitter junction with the LED acting as the base of a transistor, so if the forward current through the LED is 1mA then the collector current will be 1.6mA (assuming the device is not saturated). The LED on the opto can handle a reverse voltage of 6V, so the user needs to be careful when connecting an input greater than that and make sure to not connect it backwards.

The collector of the opto is connected to 5V and the emitter connects to the microcontroller input pin and a 20kΩ pulldown, making the input active-high. With a typical collector-emitter saturation voltage of 0.12V (max is 0.4V), the collector current is

$$\begin{aligned} I_C &= \frac{V_{in} - V_{CESat}}{R_L} \\ &= \frac{5 - 0.12}{10k} \\ I_C &= 0.244mA \end{aligned}$$

so the LED forward current times the current transfer ratio must be greater than 0.244mA in order to drive the opto in to saturation. This is much lower than the maximum allowed collector current of 50mA.

Between the positive input pin and the anode of the LED is a 3.3kΩ current-limiting resistor. An LCD controller board with a 5V backlight enable signal will, with a typical LED forward voltage of 1.1V, need to be capable of supplying

$$\begin{aligned} I_F &= \frac{V_{in} - V_f}{R_I} \\ &= \frac{5 - 1.1}{3.3k} \\ I_F &= 1.18mA \end{aligned}$$

of current. This is very close to the test current at which the opto's transfer ratio is rated. Using an ambient temperature of 40℃ derates the ratio to 90%, according to Figure 9 on page 5 of the datasheet, giving a minimum ratio of 1.44. This make the theoretical collector current 1.70mA, driving the opto well into saturation. Performing similar math for a 12V input shows that it will need to be able to supply about 3.3mA of current and will provide a theoretical collector current of 4.75mA. Doing the same math, but with V_F at the max of 1.6V, shows that a 5V backlight signal will supply about 1mA with a theoretical collector current of 1.44mA and a 12V signal will supply 3.15mA with a collector current of about 4.53mA. In short, LCD controller boards that provide a 5V or 12V backlight signal should have no trouble driving the opto into saturation and should be able to do so while supplying only a small amount of current.

Future LCD controller boards, however, could have a 3.3V or lower backlight control signal. Doing the above math again with a 3.3V signal yields an LED forward current of

$$
\begin{aligned}
I_F &= \frac{V_{in} - V_f}{R_I} \\
&= \frac{3.3 - 1.1}{3.3k} \\
I_F &= 0.67mA
\end{aligned}
$$

when using the typical LED forward voltage rating. Because the current is below the 1mA test current used by the datasheet, a additional derating of the current transfer ratio needs to be applied. Figure 11 on page 5 of the datasheet shows a derating of about 75% at 25℃ ambient. Applying that to the already-derated current transfer ratio of 1.44 gives 1.08. This means that a 3.3V input will typically have a theoretical collector current of 0.72mA. This is 3x higher than the actual collector current, which should easily drive the opto into saturation. Doing the same math with the max forward voltage of 1.6V gives a forward current of 0.5mA for a collector current of 0.54mA. This is a bit over twice the actual collector current, so the opto should still be driven into saturation.

It should be noted that orignially a more standard zener diode circuit was considered. However, zener diodes appeared to be rated with a 5mA current. This would mean that more current is required from LCD controller boards that output their backlight signals to this board. Drive specificatons are rarely, if ever, available for LCD controller boards so it was decided to aim for a lower current draw, which the VOM618A allows since its test current is 1mA.

2.3.10 Mounting Holes

The Projector Controller will have one mounting hole on each of its four corners. The holes have a 4mm inner diameter and 10mm outer diameter for screw head clearance. This is the same size used for PC motherboards, so standoffs and screws that come with PC cases should be useable with this board. The holes will accommodate either M3 metric screws or 6-32 standard UTS screws.

2.4 Power Usage

The following tables lists the current draw of the different components on the board. When possible, the values used are the worst-case useage given in the data sheet; actual usage should normally be lower. The device has 5V and 12V power, so multiplying the current given in the table by its voltage input will give the power used.

Component	Current (mA)	Notes
5V Components		
ATMega328P Active Current	23	See Note below
ECS CSM-3X 20MHz Crystal	0.02	Derived from "Drive Level" rating
FT232R Operating Current	15	Powered only from USB VBUS
FT232R USB_PWREN Current	0.5	Powered only from USB VBUS
12V Feedback Divider	0.9	R1 = 10kΩ, R2 = 3.3kΩ
TMP36 Supply Current	0.05	
BSP772T Input Current	0.015	Current at IN pin while device is on
MCP3301 V_{REF} Current	0.15	Current at V_{REF} while clocking data
MCP3301 Operating Current	0.45	
MCP3301 2.5V Divider	0.25	R1 = R2 = 10kΩ
Thermistor Input Current	0.5	Worst case is shorted thermistor
VOM618A-4T Collector Current	0.25	Assumes $V_{CE} = 0$
Green LED	6.1	
Yellow LED	2	
Red LED	1.3	
5V, 10kΩ Pulldowns (4)	0.5 x4	
5V, 10kΩ Pullups (2)	0.5 x2	
Total	**53.485**	
12V Components		
5V Component Draw	53.485	See above
L78M05C Quiescent Current	6	
BSP762T Operating Current	1.3	
Total	**60.785**	

Note: Values were taken from Section 30.7 of the datasheet and includes all peripherals and programming current but not external IO. This assumes a 5.0V input voltage and 20MHz clock input.

Any external 5V devices connected to the Projector Controller, including a board-powered ISP programmer, must keep the total current draw under 100mA. Given the total shown above, external devices can use up to 46mA or so.

In addition to the loads shown here, the 12V supply also needs to be able to handle the external fan and 4 outputs. Therefore, the user will need to determine the current draw of his or her choosen components and size a power supply that can handle those plus another 61mA for this board.

2.5 Microcontroller Pinout

Following is a table showing how the pins on the microcontroller are used and their net names on the schematic, if applicable.

Pin #	Pin Name	Net Name	Description
1	(PCINT14/$\overline{\text{RESET}}$) PC6	$\overline{\text{RESET}}$	Active-low CPU reset line
2	(PCINT16/RXD) PD0	UART_RX	UART receiver input
3	(PCINT17/TXD) PD1	UART_TX	UART transmitter output
4	(PCINT18/INT0) PD2	FAN_EN	Fan power enable
5	(PCINT19/OC2B/INT1) PD3	FAN_PWM	25kHz fan control PWM signal
6	(PCINT20/XCK/T0) PD4	⟨None⟩	Bootloader enable pin
7	VCC	⟨None⟩	Main supply input
8	GND	⟨None⟩	Ground
9	(PCINT6/XTAL1/TOSC1) PB6	⟨None⟩	20MHz crystal oscillator input
10	(PCINT7/XTAL2/TOSC2) PB7	⟨None⟩	20MHz crystal oscillator input
11	(PCINT21/OC0B/T1) PD5	LEN_EN_RED	Red LED enable
12	(PCINT22/OC0A/AIN0) PD6	LED_EN_YELLOW	Yellow LED enable
13	(PCINT23/AIN1) PD7	LED_EN_GREEN	Green LED enable
14	(PCINT0/CLKO/ICP1) PB0	LAMP_EN	Lamp relay enable
15	(PCINT1/OC1A) PB1	⟨Unused⟩	Not used
16	(PCINT2/OC1B/SS) PB2	$\overline{\text{ADC\_CS}}$	SPI ADC chip select
17	(PCINT3/OC2A/MOSI) PB3	MOSI	SPI master output
18	(PCINT4/MISO) PB4	MISO	SPI master input
19	(PCINT5/SCK) PB5	SCK	SPI clock
20	AVCC	⟨None⟩	Analog suppy input
21	AREF	⟨None⟩	Analog reference voltage
22	GND	⟨None⟩	Ground
23	(PCINT8/ADC0) PC0	AVA_12V_FB	12V supply feedback
24	(PCINT9/ADC1) PC1	ANA_AMBIENT	Ambient temperature reading
25	(PCINT10/ADC2) PC2	PROJECTOR_ON	Projector on input
26	(PCINT11/ADC3) PC3	⟨Unused⟩	Not used
27	(PCINT12/ADC4) PC4	⟨Unused⟩	Not used
28	(PCINT13/ADC5) PC5	⟨Unused⟩	Not used

2.6 Connector Pinout

The following tables gives the functions of the pins on each connector. Pin 1 on through-hole connectors can be found by looking for the square solder pad around the through-hole. The USB connector is the only surface mount connector and pin 1 is denoted by a small circle next to the pin.

CON1: Communication (USB Mini-B)

Pin #	Description
1	VBUS
2	D-
3	D+
4	ID (Not Used)
5	GND

CON2: Projector On Input (Mini CT 3-pin)

Pin #	Description
1	Input positve
2	Not Used
3	Input Negative

CON3: Thermistor (JST PH 2-pin)

Pin #	Description
1	5V to thermistor
2	Voltage from thermistor to ADC

CON4: 12V Input (Screw Terminal)

Pin #	Description
1	12V
2	Ground

CON5: Fan Output (4-pin PC Fan Connector)

Pin #	Description
1	Ground
2	12V
3	Sense (Not Used)
4	Control (PWM)

CON6: External Red LED (Mini CT 2-pin)

Pin #	Description
1	12V
2	Ground

CON7: External Yellow LED (Mini CT 2-pin)

Pin #	Description
1	12V
2	Ground

CON8: AVR ISP Header (10-pin Shrouded)

Pin #	Description
1	MOSI (SPI Out)
2	VCC (5V)
3	LED/NC (Not Used)
4	Ground
5	Reset
6	Ground
7	SCK (SPI Clock)
8	Ground
9	MISO (SPI In)
10	Ground

CON9: External Green LED (Mini CT 2-pin)

Pin #	Description
1	12V
2	Ground

CON10: Lamp Relay Enable (Mini CT 2-pin)

Pin #	Description
1	12V
2	Ground

JP1: Bootloader Enable Jumper

Pin #	Description
1	Jumper Pin 1
2	Jumper Pin 2

2.7 Conversion Formulas

The conversion formulas for the analong inputs discussed earlier are available here as well for convenience.

2.7.1 12V Feedback

$$V_{ADC} = \frac{5 * Counts}{1024}$$
$$V_{FB} = V_{ADC} * 4.03$$

2.7.2 Ambient Temperature

$$V_{ADC} = \frac{5 * Counts}{1024}$$
$$T_{Amb} = 100 * (V_{ADC} - 0.5)$$

2.7.3 Thermistor Temperature

The MCP3301 is a signed 13-bit ADC, so the reading must be sign-extended first. Notice that a reading of -4096 will cause a divide by zero error with these formulas. This indicates a shorted thermistor and so is an abnormal condition. When calculating temperature with this conversion, the input should be clamped to ±4095.

$$V_{ADC} = \frac{2.5 * Counts}{4096} + 2.5$$
$$R = \frac{5}{V_{ADC}} - 1$$
$$T = \frac{1}{A + B\ln(R) + C\ln^2(R) + D\ln^3(R)} - 273.15$$

where

$$A = 0.0033540164$$
$$B = 0.0002565236$$
$$C = 2.605970e - 6$$
$$D = 6.329261e - 8$$

3 Firmware

The onboard ATMega328P has two different firmwares on it. The first is a small bootloader that must be programmed on using an external programming device. A 10-pin header is provided onboard for this purpose. The bootloader is reponsible for programming application code into flash, allowing field updates without the programming device. and to make sure a functional application is present before handing over control to it. The second firmware is the actual application code. This is programmed on by the bootloader and is what actually does the projector controlling.

This section will describe functionality common to each firmware and then delve into the functions specific to each. First is information on how to build and program the firmwares onto the board.

3.1 Tools Used

The firmware was developed in various versions Ubuntu Linux with the current one as of this writing being version 12.10 Quantal Quetzal. The applications listed should be available on other distributions (or they can be built from source), but they may use different package names from what is given here.

The firmwares use the AVR GCC cross-compiler, which is at version 4.7 as of this writing, with the associated LibC and Binutils packages. AVRDUDE is used to program the built firmware onto the board. Building and programming the firmware is handled by the SCons build tool, which itself requires Python 2.4 or higher.

On Ubuntu 12.10, the following packages need to be installed to get everything:

- gcc-avr
- gcc-binutils
- avr-libc
- avrdude
- scons

The `scons` package should also grab the needed Python package. If not, then that will need to be installed as well.

Windows users can use Atmel Studio, which is a free IDE provided by Atmel which also includes the needed toolchain. The IDE is available on Atmel's website at http://www.atmel.com/tools/atmelstudio.aspx?tab=overview. Using the IDE itself would require one to create projects, one each for the bootloader and application, for the IDE since they are not provided here. Another possibility is to install SCons for Windows and Python alongside the IDE and build using SCons. The SCons Windows installer is available on the SCons website at http://www.scons.org/download.php. SCons should be able to find the toolchain as long as it is in the Windows PATH.

3.2 Building With SCons

SCons is a build system (essentially a replacement for GNU Make) written in Python. The build scripts themselves are special Python files. In the `firmware/` is a file called `SConstruct`. This is the top-level build script and contains settings that are used by both the application and bootloader, such as include file paths and some compiler options. The `firmware/bootloader/` and `firmware/app/` directories each contain a file called `SConscript`. These are called from the `SConstruct` file and contain options specific to the app or bootloader, such as the output file path, source files, and so on. The files themselves are commented so that anyone who wishes to modify build options can at least figure out what needs to change in the files.

The SCons documentation is very good and even those inexperienced with SCons can set up a build just by following the documentation. The documentation is available on the SCons website at http://www.scons.org/doc/production/HTML/scons-user/index.html and a wiki is available at http://www.scons.org/wiki/.

To use SCons, open up a command-line window or terminal and navigate to the location of the `SConstruct` file, which is `firmware/`. Now run `scons ⟨command⟩`, where `⟨command⟩` is one of the following:

build-bootloader :
Build the bootloader using AVR GCC and output the resulting Intel hex file to `firmware/bootloader/output/pjc-bootloader.hex`.

write-bootloader :
Write the bootloader to the microcontroller using AVRDUDE and a compatible programmer (see below). The bootloader must have been built using `build-bootloader` for this to work.

read-flash :
Read the entire contents of the microcontroller's flash memory and output the contents as an Intel hex file to `firmware/bootloader/output/pjc-flash-image.hex`.

erase-device :
Erase the device's flash, EEPROM, and lock bits, but not its fuse bits.

write-fuses :
Write the proper fuse and lock bits to the device. This should be done the first time a new device is programmed.

build-app :
Build the application using AVR GCC and output the resulting Intel hex file to `firmware/app/output/pjc-app.hex`.

Multiple commands can be executed at once by putting more than one on the command line. For example, running `scons build-bootloader write-bootloader` will compile the bootloader and, if successful, program it to flash.

The actions of these commands are defined in the `SConscript` files. For example. the values of the fuse bits programmed using the `write-fuses` command are defined in the `firmware/bootloader/SConscript` file.

3.3 Programming Flash

An external programming device is required to program the bootloader onto the device. A 10-pin header is provided onboard for this purpose. The header pinout is standard across AVR device programmers and so there are multiple programmers that will work.

The build scripts are configured to use the "usbtiny" programmer protocol with AVRDUDE. Any programmer that conforms to that protocol will work. Two such low-cost examples are the Adafruit USBtinyISP (available at http://www.ladyada.net/make/usbtinyisp/) and the SparkFun Pocket AVR Programmer (available at https://www.sparkfun.com/products/9825).

To use a different programmer, open `firmware/bootloader/SConscript` and look for the line that says "`icspdevice = 'usbtiny'`". Change it to a programmer that AVRDUDE recognizes (find AVRDUDE documentation online) and save the file.

Linux users may find that trying to program a device results in AVRDUDE complaining that it cannot talk to the programmer, saying something along the lines of "Operation not permitted". If this happens, running the command as root should make it work. Another option is to create a new udev rule for the programmer. Create a new file in `/etc/udev/rules.d/`, or wherever udev rules are stored, and add the line

```
SUBSYSTEM=="usb", ATTRidProduct=="0c9f", ATTRidVendor=="1781", MODE="0666", GROUP="adm"
```

Save the file with a descriptive name, such as `90-avrisp.rules`. The number determines the order in which the rule is processed. Higher-numbered rules override lower-numbered rules. Disconnect and reconnect the programmer and try again. Note that the "idProduct" (0c9f) and "idVendor" (1781) are dependent on the actual device; the values shown are for the "usbtiny" programmers mentioned earlier. Use the `lsusb` command to get the correct values for different devices.

3.4 Fuse and Lock Bits

The AVR microcontroller have special locations in flash memory that hold configuration registers for the device. These control hardware configuration options such as the size of bootloader space, the clock source, and write-protection features. The mapping for the fuse and lock registers is given in Section 28 of the ATMega's datasheet. The following table summarizes the fuse bit configuration used by the Projector Controller.

Register	Value	Notes
lfuse	0xF7	Use external full-swing crystal and longest startup time
hfuse	0xD8	Defaults except set to use bootloader reset vector; that is, start in the bootloader rather than application
efuse	0x04	Brown-out detection enabled at 4.3V
lock	0x3F	Defaults; no lock features enabled

The fuse and lock bit values programmed onto the device are stored in `firmware/bootloader/SConscript`. Use caution when changing these as the wrong values can make the device unprogrammable. In that case, one would need to purchase a parallel programmer to correct the problem. Doing a search for "AVR fuse calculator" online will bring up utilities that provide help in setting the bits to the correct values.

3.5 Intel Hex Files

The firmwares are stored as Intel hex files, which are text files that store binary information, such as program data for the firmware. Each line is a record which can contain program data, address information, or an indicator that the end of file was reached. The file is parsed one record at a time to ultimately yield a binary image of the data. Below is a typical record.

```
:10700000CEC00000F3C00000F1C00000EFC00000DF
```

The colon (":") at the beginning indicates the start of the record. Every record has this and it does not affect the data within.

The first two hex digits give the number of bytes in the data field. In this record, there are 0x10 (16 decimal) data bytes. There are always 2 digits.

The next four digits are the lower 16-bits of the address for the start of the data. The upper 16-bits of the address, if needed, are set by different record types. This field is used only for Data Records, which is what this record is. Other record types leave this as 0. Note that there are always 4 digits. The address is big endian, meaning that the most significant byte comes first.

Next up are 2 digits that give the record type. There are six records types.

00, Data Record
: This type holds data that is part of the firmware and so would be programmed into flash. The 16-bit address would be added to the upper portion given by record types 02 or 04 to get the full address, if needed.

01, End Of File Record
Indicates that the end of the file has been reached. This is the last record in the file and usually looks like `:00000001FF`. It contains no data.

02, Extended Segment Address Record
The data field of this record contains two bytes of data that together for the upper 16 bits of a 20-bit address. The address is big endian. Shift the data left by 4 bits to get the actual upper portion of the address.

03, Start Segment Address Record
This record is not relevant to the AVR architecture and so is not used.

04, Extended Linear Address Record
Simiar to record 02, except the data forms the upper 16 bits of a 32-bit address. Shift the data left by 16 bits to get the upper portion of the address. The address is big endian.

05, Start Linear Address Record
This record is not relevant to the AVR architecture and so is not used.

The next field is the data field and its size is given by the size field at the start of the record. Its contents depeds on the record type. In this case, the data field contains 16 bytes of program data since this is a Data Record.

The final field is a checksum. This is the two's complement of the sum of all of the other bytes in the record. To determine the checksum, add up the values of the bytes in the other records, XOR the sum with 0xFF, and add 0x01. The checksum is the lowest byte of that result. To verify a checksum, add up all of the bytes, including the checksum, and check that the result is 0. If so, then the checksum is good.

Software that wants to update the Projector Controller's application through the bootloader will need to be able to parse Intel hex files. The file `/software/flashimge.py` contains code that parses the hex file and can be used as a reference.

3.6 Linker Scripts

The bootloader and application directories contain linker scripts `pjcbootloader.x` and `pjcapp.x`. The linker scripts serve two purposes.

The first purpose is to partition the flash memory betweent the application and bootloader so that they do not collide. The ATMega328P has hardware support for a bootloader, allowing one to use fuse bits to allocate a portion of the end of flash to a bootloader. The Projector Controller uses the last 4KB of flash for the bootloader and so the linker script is set up so that all bootloader program data goes into that section of flash. The application's linker script allocates the lower 28KB of flash to the application.

The second purpose is to allow the application to jump back into the bootloader by simply clearing a flag and performing a soft reset. The linker scripts define a special NOLOAD section called ".shareddata" that contains a single boolean variable. A NOLOAD section is one that is not initialized during program startup, meaning that the data within it will survive a software reset. The bootloader can check the state of the flag to see if it was cleared. If it was, or if the CPU was not reset by software, the bootloader will start the app if able. Otherwise, it will stay in the bootloader.

Firmware support for these linker script features is implemented in `sharedmem.c` and `sharedmem.h` in `firmware/common/`. The files are commented to help explain what they are doing.

Performing an online search for "ld linker scripts" will produce information about how to use linker scripts. One such manual can be found at http://sourceware.org/binutils/docs-2.22/ld/Scripts.html.

3.7 Watchdog

The firmwares make use of the ATMega's built-in watchdog timer. The watchdog is a timer that must be periodically cleared. If the watchdog times out before it is cleared, it will reset the device. The idea is that this will prevent the device from getting stuck in a never-ending loop since the watchdog will reset it. Both firmwares use a watchdog timeout of 250ms.

Also, because the AVR instruction set does not have a `reset` instruction, the watchdog is used to perform a soft reset. To reset the device, the watchdog is set to its shortest timeout of 15ms and the firmware loops until the watchdog trips.

3.8 LEDs

The firmwares use the onboard green, yellow, and red LEDs to communicate basic information to the outside world. If no LEDs are lit, then either the board is not powered on or the micrcontroller is blank (that is, no firmware is on it).

If only the yellow LED is on, then the bootloader is running. The bootloader does not use the other two LEDs.

The green LED is on if the application is running. The following table gives the meanings of the possible LED combinations used by the firmwares, including the ones listed above.

Green	Yellow	Red	Meaning
Off	Off	Off	Board not powered or no bootloader or app programmed
Off	On	Off	Bootloader is running
On	Off	Off	App is running, but not doing anything
On	On	Off	App is running; lamp and fan are on
On	Blink	Off	App is running; lamp is off and fan is on
On	Off	On	App encountered error, such as over-temp or missing thermistor
On	Blink	On	App encountered error and fan is on
On	On	On	Not used; should never see this

3.9 Register Map

The file `firmware/common/regmap.h` defines bit fields for most of the ATMega328P registers, making accessing them easier. Registers that do not have any bit fields in them are not defined in this file. All of the registers defined here follow a simple naming convention:

```
REGNAMEbits.field
```

where `REGNAME` is the name of the register, such as `PORTB`, and `field` is the lower-case name of a field in the register, such as `pb1`. In this case, the full qualified name would be `PORTBbits.pb1`. The register can be read from and written to in this manner. The addresses of the registers are known at compile time, meaning that the compiler can use the most efficient AVR instruction to access the register.

Register fields that are multiple bits wide can be accessed either as a whole field or by individual bit. For example the `SMCR` register, used for controlling sleep modes, has the fields `SM0`, `SM1`, and `SM2` which could be treated as a single 3-bit field. The whole field can accessed with `SMCRbits.sm` or a single bit accessed with `SMCRbits.sm1`, for example.

Section 31 of the ATMega328P datasheet has a list of all of the device's registers.

3.10 Flash and EEPROM

The ATMega328P has 32KB of flash memory, divded into 128-byte pages. The top 4KB of flash are configured for use by the bootloader, making the lower 28KB available to an application. Since each page is 128 bytes, there are 224 pages avilable for the application. Flash can be programmed one byte at a time, but must be erased one page at a time. The bootloader will always program one page at a time and erase the entire 28KB app space.

The last page of flash is used by the bootloader to store the application's 16-bit CRC and a key that is present when an app is programmed. The bootloader uses these to verify that there is an application onboard and that it is valid.

The microcontroller also has 1KB of EEPROM that the application uses to store setting that need to be restored on start up. These include things like the minimum fan speed, over-tempearture limit, and control loop settings. The application can read and write this EEPROM space. The bootloader does not read it, but can be instructed to erase the contents of EEPROM. This is useful in the case that some bad EEPROM settings are causing the application to misbehave.

3.11 Communication

The firmwares use the ATMega328P's UART module for communicating to whatever PC it is connected to. The UART lines go into the onboard FTDI FT232R, which is a UART-to-USB translator that appears to the PC as a virtual COM port. Any software that can talk to serial ports, such as TeraTerm or the software used with the Projector Controller, can talk to the device provided the right drivers are installed. Newer operating systems should have drivers for the device pre-installed; however, drivers are also available from http://www.ftdichip.com/FTDrivers.htm.

The firmwares talk at 115200 baud using 8 bits per transfer, no parity bit, no flow control, and one stop bit (8-N-1). Carriage returns are used as line endings.

Communication is handled on two levels. The first level is the handling of the actual UART interface on the device and this is done in `firmware/common/uart.c` and its header `uart.h`. The module contains a transmit and receive buffer and uses interrupts to transmit data from the transmit buffer and receive data into the receive buffer. The transmit and receive functions in the module access those buffers and can transmit single bytes, byte arrays, or strings.

The default size of each UART buffer is 64 bytes. Any software talking to the firmwares is responsible for not overflowing the buffer.

The second level is the command inteface in `firmware/common/cmd.c` and its header `cmd.h`. This module implements a state machine to run a command prompt interface that can be used in a program like TeraTerm or HyperTerminal. The state machine receives data from the UART module until either it see a carriage return or until the command buffer is full. It will also look for backspaces to remove the most recent character and the Escape key to clear the buffer entirely. When a new command string is received, it assumes that everything until the first space or carriage return is the name of the command. It looks through its list of registered commands for a match and, if one is found, calls the function associated with that command. If no match is found, the module outputs "Unknown command" to indicate such.

Command are registered via the `Cmd_RegisterCommand` function. This function takes the name of the command, a handler function pointer, and a short help string for the command. The handler function signature must match `void func(const char*, uint8_t)`, where the parameters are the command string and its length. The handler function would parse the command string as needed, execute its command, and send out a response. The module has a few common commands defined, which can be used as examples.

The default size of the command buffer is 32 bytes plus one for a null terminator. The default maximum number of commands is 10.

At startup, the module will transmit the string "`---Startup---`" (without quotes) and a version string. The

former string allows software to determine when the device restarted when it should not have.

3.12 Firmware Config

Both the application and bootloader have a file called `appcfg.h`. This file defines compile time configuration settings for the firmware such as baud rate or the maximum number of serial commands. The header files for some of the modules in `firmware/common` will define certain macros if they have not been defined already. These are what control baud rate, buffer sizes, and the like. To use non-default values, simply define the macros in the `appcfg.h` files with the desired value. As an example, take a look at the bootloader's config file `firmware/bootloader/appcfg.h` as the bootloader uses a couple of non-default values.

Also defined in these files are a version string and version number, which is sent over the UART when the device starts up or in reaction to the "v" command. The version string for the bootloader should contain the word "Bootloader" (case sensitive) in it; likewise, the application should *not* have the word "Bootloader" in it. The Projector Controller software looks for that word to determine if the device is running the bootloader. The bootloader's `appcfg.h` also defines the macro `PJC_BOOTLOADER__`, which allows common code to act differently if the bootloader is being built. The application should not define this macro.

3.13 Bootloader Specific

The bootloader firmware is the first to run when the microcontroller powers up and is responsible for making sure that an application is on board and jumping to it. The bootloader can also be told to stay there instead of jumping to the application firmware, allowing one to perform an application update in the field without a programmer.

3.13.1 Version Changes

This section describes changes made to the different release versions of firmware. Minor updates commited only to the repository and not released for public download may not be shown here. In that case, take a look at the repository logs to see what has changed.

Revision	Date	Changes
0	Feb. 14, 2013	Initial revision

3.13.2 Bootloader Startup

When the bootloader starts up, it goes through a series of checks to see whether it should start the application or stay in the bootloader.

If any checks turn up something that suggest the bootloader should not start the application, the bootloader will turn on the yellow LED and print a message to the serial console stating the reason why the bootloader is running. The message is formatted as "(N) Text", where 'N' is one of the following numbers corresonding to a check and "Text" is a textual explanation of the check. This format allows software to simply look for the number rather than having to parse the string to determine why the bootloader is running. The number is always 2 digits, so an example message would look like "(01) Bootloader pin set". The message is followed by a carriage return.

1. **Is the bootloader pin set?**
   The board has a jumper on it going into pin 6 (PD4) that can be shorted to tell the bootloader to not jump to the applcation. This is useful for recovering from an application that will not start correctly.
2. **Did the watchdog reset the application?**
   If the watchdog resets the device, it should end up back in the bootloader so that the application is not

stuck in a reboot loop. The application can also purposefully reset the CPU by waiting for the watchdog to timeout. See the Watchdog section for more info.

3. **Is there no application on the board?**
   The bootloader stores the application's CRC and a key indicating that an application is programmed onto the device. If the key is not present or incorrect, the bootloader will assume that no valid application is loaded.
4. **Is the application's CRC bad?**
   The bootloader calculates the CRC of the application data stored in flash memory and compares that to the CRC stored in flash (which is calculated when the application is updated). If the CRCs do not match, the bootloader will not start the application.

Issuing the "j" command over the serial interface will tell the bootloader to jump to the appication, ignoring the above checks. The "s" command will reprint the status message described above.

If the bootloader does not start up the application, it will instead set up the UART module and the command interface. The bootloader will also enable the watchdog timer and enable interrupts.

3.13.3 Bootloader Operation

If the bootloader does not jump to the application, it simply waits for incoming serial commands and acts on them.

3.13.4 Serial Commands

Below is a list of serial commands the bootloader recognizes. See the Communication section for more information on how to talk to the bootloader. The "Syntax" column shows how a valid command string is formed along with its arguments. The command and it arguments are separated by a single space between each one.

Command	Name	Syntax	Description
crc	App CRC	crc	Calculate the 16-bit for the application and print it as a four-digit hexadecimal number.
h	Help	h	Print of list of these commands with short help strings.
v	Version	v	Print firmware version information.
j	Jump	j	Jump to application.
r	Reset	r	Reset device and stay in bootloader.
ea	Erase App	ea yes	Erase the application from flash. The first argument must be the word "yes" as a confirmation of the action. Prints a status code to indicate success.
ee	Erase EEPROM	ee yes	Erase the onboard EEPROM. The first argument must be the word "yes" as a confirmation of the action. Prints a status code to indicate success.
pd	Page Data	pd *offset checksum data*	Write *data* to a temporary page buffer, where it is stored before being programmed to flash with the "pp" command. The *offset* is the starting position in the buffer in which to put the *data*. The *checksum* is calculated by adding all of the bytes in *data*. Prints a status code to indicate success.

Command	Name	Syntax	Description
pp	Program Page	pp *pagenum*	Program the flash page given by *pagenum* with the data stored using the "pd" command. The *pagenum* parameter can range from 0 to 223 for the ATMega328P. The contents of the page are verified by comparing it to the data stored in the page buffer to make sure the flash ware written correctly. Prints a status code to indicate success.
wc	Write CRC	wc	Calculate the application's CRC and write it to a special spot in flash. Do this after all of the application pages were successfully written to complete the update. The bootloader will calculate the application's CRC on startup and compare it to this value to make sure the app is valid before jumping to it. Prints a status code to indicate success.
ps	Page Size	ps	Print the size of a single flash page, which is 128 on the ATMega328P. This is useful mainly for porting the bootloader to other devices. Software can query this value to know how much data to load with the "pd" command. The value is printed as 4 hexadecimal digits.
pn	Number of Pages	pn	Print the number of flash pages availabe to the application, which is 224 on the ATMega328P. This is useful mainly for porting the bootloader to other devices. Software can query this value to know how many pages of data it must be able to handle. The value is printed as 4 hexadecimal digits.
s	Boot Status	s	Reprint the status message that appears when the bootloader starts up. See the Bootloader Startup section for more info.

The first three commands are common to the bootloader and application and so are implemented in `firmware/common/cmd.c`. The rest are bootloader-specific and are implemented in `firmware/bootloader/localcmd.c`.

Some of these commands return a status code to indicate what happened with the command. The possible codes are as follows. The codes are printed as "!00" (exclamation point and two hex digits) followed by a carriage return.

Code	Name	Description
00	OK	No error; this is returned when the command executes successfully.
01	Argument Error	The command sent to the bootloader is missing a required argument. Verify that the arguments sent down are correct (see the list above) and they are are separated by a space.
02	Range Error	An argument is out of range and so is not valid. This can happen, for example, when trying to write page to an offset outside of the page or when trying to program a page that is not in app space.
03	Mismatch Error	The command executed, but could not complete because an expected value was not found. This can happen when a flash write fails or when a page checksum does not match the data.

3.13.5 Performing an Update

A firmware update is performed using the serial commands given in the Serial Commands section above. In essence, the software talking to the bootloader to do the update must parse the application's Intel hex file into

a binary image of the micontroller's memory and send that down to the bootloader in chunks to program the device. When all of the data is sent, the software will finalize the update by telling the bootloader to calculate and write the application's CRC to a special location at the end of flash.

Following is a basic outline of how to perform an update. Anyone trying to write software to do this will need to be familiar with the Intel hex file format. There are many sources available online that can help with this and the Intel Hex Files section above may be of use. This section assumes that the ATMega328P is being used, though one could conceivably port the bootloader to a different device and follow a similar procedure.

1. Make sure the microcontroller is running the bootloader. To do this, run the "v" command to get version information and look for the word "Bootloader" (case sensitive). If it is present, move to the next step. If not, jump to it using the "j" command. The version is printed during startup, so look for "Bootloader" again to see if the jump succeeded.
2. Create a block of memory the size of the microcontroller's flash. The "ps" and "pn" commands are useful for doing this generically. The page size of the ATMega328P is 128 bytes and it has 224 pages available to the application for 28KB total. This will be referred to as the "flash image". Initialize this so that every byte is 0xFF.
3. Open the application's Intel hex file. For each line, parse the record and build up the data for the flash image. Remember that Data Records (type 00) are the only ones that contain actual flash data and that the Extended Segment Address Record (type 02) and Extended Linear Address Record (type 04) contain the upper portion of the flash address. Be sure the validate the record's checksum and to make sure the address is within the app space (this is 0 to 0x6FFF for the ATMega328P).
4. Erase the old application using the "ea yes" command. The "yes" argument is required as a confirmation of the action. It will return when the erase is complete, which may take a few seconds. Move on if successful.
5. Begin sending data to the bootloader. Read the first 16 bytes from the flash image and send them using the "pd" command. In this case, the *offset* will be 0. Add those 16 bytes together to get the *checksum*. The *data* for the command is those 16 bytes concatenated into a string of their hex digits. For example, if the data read from the flash image was {0x00, 0x11, 0x22,..., 0xEE, 0xFF}, then the string would look like "001122...EEFF". Send the command down with the arguments and look for the status code coming back. If the status code indicates a problem, verify that the command was sent correctly and that the checksum is correct.
6. Do the previous step 7 more times, increasing the *offset* by 16 and reading the next 16 bytes from the flash image each time. This will have sent 128 bytes–a single page's worth–of data to the bootloader.
7. Now that enough data was written to the bootloader to fill a page, program the page using the "pp" command. The *pagenum* parameter is 0 in this case since this is the first page being written. Look for the status the bootloader prints to make sure the command executed successfully.
8. Perform the previous three steps repeatedly while reading the next 16 bytes from the flash image, starting the *offset* back at 0, and incrementing the *pagenum* by 1 each time, until all of the flash data has been programmed. Note that it is not necessary to program pages that have no data in them (all bytes are 0xFF). The software can stop short if it knows where the program would end. One way to do this is to find the last byte that is not equal to 0xFF and make the containing page the last one to be programmed.
9. If the programming operations completed successfully, tell the bootloader to calculate a CRC for the application and write it to flash by sending it the "wc" command. If the status code returned by it indicates success, then the update is complete.
10. If the update has completed successfully, jump to the newly-programmed application by sending the "j" command.

The Projector Controller software follows this procedure for performing an application update. The file `software/serialcomm.py` has the function `doFirmwareUpdate` that performs the update. It makes use of the class found in `software/flashimage.py` to parse the hex file and store the data and uses the class in `software/pjcbootloader.py` to send the commands needed. One familiar with Python can use this as a reference for updating application firmware.

3.14 Application Specific

The application is what performs the actual controlling functions of the Projector Controller. It sets up and controls the microcontroller's IO pins and the various onboard devices. More information about these devices is available in the Hardware section. The Microcontroller Pinout section lists the uses of the pins on the ATMega328P device.

3.14.1 Version Changes

This section describes changes made to the different release versions of firmware. Minor updates commited only to the repository and not released for public download may not be shown here. In that case, take a look at the repository logs to see what has changed.

Revision	Date	Changes
0	Feb. 14, 2013	Initial revision

3.14.2 Application Startup

The following steps are performed when the application starts up in order to set up the needed modules for operation.

1. Light green LED to indicate the application is running.
2. Start one-millisecond free-running timer.
3. Set used digital IO pins to their proper input or ouput states.
4. Initialize SPI module used to talk to MCP3301.
5. Set up analog inputs and start conversion.
6. Enable 250ms watchdog timer.
7. Initialize UART module.
8. Initialize command interface and register application commands.
9. Enable interrupts.

None of the external devices require an explicit setup sequence and so are immediately usable assuming the IO ports on the microcontroller used to communicate with them are set up correctly.

3.14.3 Application Operation

The main task of the application is to control the fan speed while the lamp is running and shut the lamp down if the temperature of the thermistor exceeds the programmed limit. This is done by running a state machine on a regular interval in which the state depends on the state of the lamp and fan. See the State Machine section for more information. This state machine controls the yellow and red LEDs.

The application will also monitor the command interface for new commands as well as monitor the digital and analog inputs. The ADCs are read at 5Hz and the digital input is monitored once every 10ms.

3.14.4 Free-Running Timer

The firmware uses Timer 0 to create a free-running millisecond timer. Timer 0 is an 8-bit timer and so does not have enough range to actually acheive a one-millisecond period. Instead, it is set up for a 100µs period. The

timer will fire an interrupt when the period ends. After the interrupt fires 10 times, a counter will be incremented to indicate one millisecond of time. This interrupt routine is provided in `firmware/common/freetimer.c` and `freetimer.h` along with a delay routine and macros to check if a given amount of time has expired.

3.14.5 Serial Commands

Below is a list of serial commands the application recognizes. See the Communication section for more information on how to talk to the bootloader. The "Syntax" column shows how a valid command string is formed along with its arguments. The command and it arguments are separated by a single space between each one.

Command arguments are optional. The commands with arguments will set some value if the argument is present; otherwise, they will print the current value of the command without changing anything. Commands that set values will not save the values; use the "sv" command to do that.

Command	Name	Syntax	Description
crc	App CRC	crc	Calculate the 16-bit for the application and print is as a four-digit hexadecimal number.
h	Help	h	Print of list of these commands with short help strings.
v	Version	v	Print firmware version information.
j	Jump	j	Jump to bootloader by resetting device.
r	Reset	r	Reset device and restart application.
a	ADCs	a	Print a list of ADC readings with one reading per line. Each reading is a four-digit hexadecimal number. The order of the readings are: 12V input feedback, ambient temperature sensor, and thermistor reading (this is sign-extended).
dc	Duty Cycle	dc	Print the current fan duty cycle, which is proportional to its speed, as a float.
dcl	Duty Cycle Low	dcl *dutycycle*	Set the minimum duty cycle in percent for the PWM controlling the fan speed and print that as a floating point value. Range is 0 to 100%, but must be lower than or equal to the Duty Cycle High value. Setting this below 30% is not recommended since the fan may not run at such a low duty cycle.
dch	Duty Cycle High	dch *dutycycle*	Set the maximum duty cycle in percent for the PWM controlling the fan speed and print that as a floating point value. Range is 0 to 100%, but must be higher than or equal to the Duty Cycle Low value. Setting this below 30% is not recommended since the fan may not run at such a low duty cycle.
le	Lamp Enable	le *en*	Enable or disable the lamp and print a non-zero decimal value if the lamp is enabled or 0 otherwise. The state of the lamp will be overriden by the Projector On input, so this is only useful for testing. This also starts the control loop and fan control.
lod	Lamp Off Delay	lod *delay*	Set the time in seconds the application will keep the lamp on after being told to turn it off and print the delay value as a float. Some LCD controllers disable their backlight while switching inputs. This command allows users to overcome that behavior by waiting to see if the LCD controller turns the lamp back on.

Command	Name	Syntax	Description
ft	Fan Off Temperature	ft *offset*	Set the temperature, relative to ambient, at which the fan will turn off after cooling the lamp down. Value is in degrees Celsius and is printed as a float. As an example, if this value is 5.0 and the ambient temperature is 27℃, then the fan would shut off at 32℃. This takes effect only when the lamp was turned off; the fan is always on when the lamp is on. Range is 0 to 125℃.
ot	Over-temp Limit	ot *limit*	Set the temperature in degrees Celsius at which the lamp will shut down due to an over-temp condition and print the new limit as a float. Range is 0 to 125℃.
tt	Target Temperature	tt *target*	Set the target temperature in degrees Celsius, which is the temperature the application will try to maintain by controlling fan speed. The closer the actual temperature is to this value, the slower the fan will spin. Prints the target temperature as a float. Range is 0 to 125℃.
er	Print Error	er *clear*	Print the most recent error code from the device as a hexadecimal value. See the Error Codes section for a list of possible return values. If *clear* is present and non-zero, the error code will be cleared so that future calls do not return the same code.
sv	Save Settings	sv	Save settings to the built-in EEPROM memory. See the EEPROM Storage section to see what is stored there.
clp	CLoop P-Gain	clp *gain*	Set the fan control loop's proportional gain value and print the value as a float.
cli	CLoop I-Gain	cli *gain*	Set the fan control loop's integral gain value and print the value as a float.
cld	CLoop D-Gain	cld *gain*	Set the fan control loop's derivative gain value and print the value as a float.
cls	CLoop I-Seed	cls *seed*	Set the fan control loop's integral seed and print the value as a float. The integral seed is the initial value of the integral tank.

Software should read back the printed value even when setting a value. This is because some commands have limits they will enforce on the input. For example, the minimum duty cycle can be in the range 0 to 100%. Trying to set it to, for example, 200% will return 100%.

The first three commands are common to the bootloader and application and so are implemented in `firmware/common/cmd.c`. The rest are application-specific and are implemented in `firmware/app/localcmd.c`.

3.14.6 Error Codes

The application will keep track of the most recent error that occurred and turn on the red LED when it does. The error is stored as a hexadecimal value and is retrievable with the "er" command. If a numerical argument is given to the "er" command and that argument is non-zero, then the error code will be cleared after the command executes. Clearing the error code will prevent it from being reported with future calls unless the error condition goes away and returns again later. The red LED will remain lit until the error condition goes away, at which point the LED will turn off.

The "er" command will print the error as a two-digit hexadecimal value, such as "00".

Here is a list of error codes the application can print. In order to make a distinction between application and bootloader error codes, application errors will start at 0x80. The exception is the code 0x00, which indicates no error.

Code	Name	Description
00	OK	No error was recorded since startup or since the last error was cleared.
80	Over-temp	The thermistor temperature was higher than the stored temperature limit (use the "ot" command to get and set this). The lamp will be shut down immediately and the fan will continue to run as normal.
81	Thermistor Short	The thermistor input is shorted and so the application cannot determine the actual temperature it is controlling. This is sensed by the ADC reading being near positive full-scale. If the lamp was on, it will be shut down immediately and the fan will run at 50% duty cycle for 30 seconds and then shut off.
82	Thermistor Open	The thermistor input is open, which could mean that no thermistor is connected. This means that the application cannot read the correct temperature. This is sensed by the ADC reading being near negative full-scale. If the lamp was on, it will be shut down immediately and the fan will run at 50% duty cycle for 30 seconds and then shut off.

These errors will turn the lamp off if it is on when the error occurs. The lamp cannot be turned back on until the error condition clears. The lamp is not turned back on automatically.

3.14.7 Control Loop

A control loop runs once every 2 seconds (0.5Hz) while the fan is on. The input to the control loop is the temperature read from the thermistor and the output is a duty cycle value used to control the fan PWM. The closer the actual temperature is to the target temperature (use the "tt" command to set and get this), the more slowly the fan will spin. Fans react slowly to changes in speed since they take time to accelerate; this is why the control loop runs so slowly.

A basic control loop algorithm is implemented in `firmware/common/pidloop.c` and `pidloop.h`. The application state machine controls when it runs.

3.14.8 State Machine

The main control function of the application firmware is implemented as a state machine, in which each state represents a different stage of control. The application moves between states in reaction to external inputs, error conditions, or after a certain amount of time. Here are the states used.

Idle
The application is in this state when the lamp and fan are both off and there are no errors, meaning that nothing needs to be done.

Lamp Start
This state is entered when the lamp is first turned on. This runs the fan at 50% duty cycle for 2 seconds before starting the control loop. This gives the fan a moment to start up and provides a good starting point for control. This turns on the yellow LED.

Lamp Run
The lamp has started up and the control loop is now running. The control loop runs once every 2 seconds.

Lamp Off Delay
The lamp was told to shut off either from the serial console or via the Projector On input. This will wait for the delay time set by the "lod" command before actually turning the lamp off. The lamp can be turned back on in this state, in which case the state machine will go back to the Lamp Run state.

Cooldown
The lamp has been shut down normally and the fan will remain on until the projector has cooled sufficiently. This is controlled by the "ft" command. This goes back to the Idle state when completed. This will blink the yellow LED.

Overtemp Error Init
Turns on the red LED due to an overtemp error and sets the error code, then immediately moves on to the Overtemp Cooldown state.

Overtemp Cooldown
This works just like the Cooldown state except it does not allow the lamp to be turned back on. Transitions to the normal Cooldown state when the thermistor temperature is below the over-temp limit and turns off the red LED in the process. This blinks the yellow LED.

Short Error Init
Turns on the red LED and sets the error code. If the lamp was on, it will shut the lamp off and move to the Bad Thermistor Cooldown state; otherwise, it will move to the Bad Thermistor Idle state.

Open Error Init
Turns on the red LED and sets the error code. If the lamp was on, it will shut the lamp off and move to the Bad Thermistor Cooldown state; otherwise, it will move to the Bad Thermistor Idle state.

Bad Thermistor Cooldown
Runs the fan at 50% duty cycle for 30 seconds and then moves to the Bad Thermistor Idle state. Blinks the yellow LED.

Bad Thermistor Idle
Similar to the Idle state except the lamp cannot be turned on while in this state. This state is used to wait for the thermistor error to clear before transistioning to the Idle state, at which point the red LED will be cleared.

The state machine is implemented in `firmware/app/control.c` and `control.h`.

3.14.9 Fan Control

The fan is controlled with a 25kHz PWM from pin 5 of the microcontroller, which is port PD3 and Timer 2's OC2B output. Timer 2 is a 8-bit timer with a PWM module attached. The PWM output is configured with a 8:1 prescaler and a period of 100 counts. This gives a control resolution of 1%, which should be more than adequate for this application.

The fan PWM code is implemented in `fimrware/app/fanpwm.c` and `fanpwm.h`.

3.14.10 Input Debounce

The Projector On input goes through a software debounce routine designed to account for possible noise from the input. The debounce routine essentially waits for the input to stabilize before reporting its new state to the rest of the application. The input is checked once every 10ms. The state of the input must remain the same for 10 readings (or 100ms) before that new state is reported.

The debounce algorithm is implemented in `firmware/common/debounce.c` and `debounce.h`.

3.14.11 EEPROM Storage

Most of the values settable via serial commands are stored in the microcontroller's built-in EEPROM. The EEPROM storage will start with a key to indicate that data exists in the EEPROM, a size field, and a version field to allow for new additions in future application updates.

The following values are stored in the EEPROM. If that item is not found in EEPROM, then the given default is used.

Item Name	Type	Default Value
Control Loop P-Gain	float	0.002
Control Loop I-Gain	float	0.01
Control Loop D-Gain	float	0
Control Loop I-Seed	float	0
Minimum Duty Cycle	float	30.0%
Maximum Duty Cycle	float	100.0%
Target Temperature	float	35.0℃
Overtemp Limit	float	45.0℃
Lamp Off Delay	float	0s
Fan Off Temperature	float	5℃ above ambient

4 Software

The software is a simple applet that allows the user to monitor information read from the connected Projector Controller and to change settings on it. It also has a section for performing application updates using the bootloader. The software is written in Python and so can be run on Windows, GNU/Linux, Mac OS X, and possibly other Unixes. A prebuilt binary is available for Windows; Linux and OS X users will need to go get Python and needed libaries, but this is not too hard.

This section is more focused on how the software is architected as opposed to what the various buttons and controls do. For info on the latter, see the *Users Guide*.

4.1 Tools Used

The software is written entirely in Python, which is what allows it to be cross-platform. The interface uses PySide, which is the official set of Qt bindings for Python. The communication with the board makes use of PySerial, an independent library for serial port access (the FT232R on the board acts as a Virtual COM Port).

The software was written with Python 2.7 and needs PySerial and PySide. Linux users should be able to get these from their distributions' repositories. Ubuntu users will need the following packages (as of Ubuntu 12.10):

- python
- python-serial
- python-pyside

Windows and Mac OS X users can download the needed packages online. Python 2.7 is available for both at `http://www.python.org/download/releases/2.7/`. Windows users can download either the 32-bit (x86) or, if using a 64-bit version of Windows, the 64-bit (x86-64) version.

PySide installers are available from `http://qt-project.org/wiki/PySideDownloads`. Follow the links at the top of the page for the correct operating system. Windows users just need to make sure to match the correct Python version (2.7) and architecture. That is, get 32-bit PySide for 32-bit Python and 64-bit PySide for 64-bit Python.

OS X users will need to get both PySide and the corresponding version of Qt. The important thing is to get the versions of PySide and Qt that go together. The downloads are under the headings showing the version numbers (eg. "PySide 1.1.1 / Qt 4.8"). The links at the top of the page just go to the main Qt website, so don't bother with those.

Getting PySerial is the same on both Windows and OS X. The instructions are availabe on the official PySerial site at `http://pyserial.sourceforge.net/pyserial.html`. Following the instructions under the "From Source" heading are probably the easiest since that involves just downloading the compressed file, extracting it, and running the command shown on that page. Windows users will need a program that can handle `tar.gz` files; 7-Zip works well. Mac OS X and many Linux distributions can open those files natively.

To make sure everything is installed properly, open up a terminal or console window and navigate to the `software/` directory. Run `./main.py` (Windows users may need to run `python main.py`) and see if the program starts up and can connect to the device. If not, make sure the packages are installed correctly and see if any errors are output to the console. Note that it might be necessary to run the program as root (or as Administrator in Windows) in order to open the serial port. It may also be necessary to manually type in the serial port path due to issues certain versions of PySerial seem to have with USB serial devices.

4.2 Building a Binary

Users can build a binary of the application using PyInstaller, a program that converts Python scripts into executable files, in addition to the tools above. PyInstaller claims to be able to generate executables under 32-bit and 64-bit Windows XP and later, Mac OS X 10.4 and later, and various Linux distros, as well as Solaris and AIX. At the the time of this writing, PyInstaller has been tried on Windows 7 64-bit without trouble.

Windows users who intend to distribute a binary version of the application should get the 32-bit version of Python and PySide using the steps in the previous section. This will allow the largest number of people to use the application without themselves having to build a binary for it.

The PyInstaller package and documentation can be found at `http://www.pyinstaller.org/`. The "Installing PyInstaller" and "Getting Started" sections in the documentation should be helpful, but the instructions essentially are to download and extract PyInstaller to some directory and run the `pyinstaller.py` file from that directory, giving it the path to `software/main.py`.

Windows users will also need `pywin32`, a package that adds Windows-specific extensions to Python. The package is availabe at `http://sourceforge.net/projects/pywin32/files/`. The documentation on that page explains how to get the right file.

To build the software into an executable, navigate to the PyInstaller directory and run

```
python pyinstaller.py -F -w path/to/software/main.py
```

The `-F` option packs all of the needed files into a single compressed executable. When the executable runs, it will decompress the files to a temporary directory and run them from there. The files are deleted when the program exits (unless the program is closed abnormally). The `-w` option tells PyInstaller that this is a windowed application. The documentation states that this is necessary for apps built for OS X.

4.3 Architecture

The software is a simple interface for sending commands to the connected Projector Controller and performing application firmware updates. The entry point for the software is `software/main.py`. When the application starts up, it creates the application window and an object that talks over the serial port (`software/serialcomm.py`). The latter is moved into its own thread, a `QThread` (types starting with 'Q' are a part of PySide/Qt), and the entry function continues on to show the window and then start the main interface event loop. The application exits when the user clicks the close button on the window header.

The SerialComm class defined in `software/serialcomm.py` runs in its own thread and is reponsible for sending data to and receiving data from the connected Projector Controller.

4.3.1 Signals and Slots

The software uses PySide for the user interface. PySide is the official set of Qt bindings for Python and Qt is a GUI toolkit orginally written in C++. Qt, and therefore PySide, makes use of what are called signals and slots for handling UI events. A signal is fired whenever a user interacts with the UI in a meaningful way, such as clicking on a button or changing the value in a text field. A slot is a listener for that signal and is what handles the button click, text field change, or whatever else happens.

To have a slot listen for a certain signal, the signal "connects" to the slot via a method call that takes the slot, which is just a function, as the argument. A signal can connect to multiple slots and a slot can listen for multiple signals. The Qt documentation states signals and slots are typesafe and threadsafe.

The software includes a utility class called ConnectionManager in the file `connmanager.py` that eases connecting signals to slots. Signals and slots are added to the object and given names. Any signals with the same

name as any slots are connected automatically. Multiple signals and multiple slots can have the same name. The advantage of this system is that different parts of the application do not need to know anything about each other. They can simply use a ConnectionManager object to advertise what signals and slots they have and the object will handle everything itself without requiring the different parts of the program to directly expose their data.

The user interface and the SerialComm class mentioned above communicate through signals and slots connected via the ConnectionManager. The user interface will send signals to the SerialComm object telling it to set or get data from the connected board. The SerialComm object will do that and then fire its own signal back to the UI containing the result of whatever command it ran. Command signals from the UI to the SerialComm object are named in the active voice, such as "SetTargetTemp" or "SetLampEnable". Response signals from the SerialComm object back to the UI are named in the passive voice, such as "TargetTempChanged" or "LampEnableChanged". The exception to this is the "WriteToLog" signal, which is a command from the SerialComm object telling the UI to print a message to the Log at the bottom of the window.

4.3.2 User Interface

The user interface is defined across four source files. The `software/mainwindow.py` file contains the top-level window. The top-level window contains controls for selecting and opening a serial port, a tab control with three pages, and a message log at the bottom. The three pages in the tab control are defined in the files files `monitorpage.py`, `settingpage.py`, and `updatepage.py`

The MainWindow class is instantiated in `main.py` and is reponsible for instantiating the rest of the interface, including the three tab control pages. The combo box at the top contains the available serial ports, which are enumerated on startup in `main.py` using PySerial. Note that, depending on PySerial version and operating system, PySerial may not find all of the available serial ports. This is mainly an issue for USB serial ports, but the correct port can be entered into the combo box manually. The log page at the bottom can hold up to 1024 entries, at which point it will being discarding old entries in favor of new ones.

It is worth noting here that the MainWindow class and the Page classes define their own signals. These signals are emitted in reaction to certain signals from controls and pass extra information that would not be passed with the control's signal. For example, the MainWindow class defines a signal called `serialopenclicked` that carries a string as its data. The "Open" button has a `clicked` signal that is connected to the `openSerial` slot (which is just a method) inside of the MainWindow class. When the "Open" button is clicked, the emitted signal activates that slot, which in turn emits the `serialopenclicked` signal. The `serialopenclicked` signal carries the currently-selected serial port as a string, passing information that is not available with just the "Open" button's `clicked` signal. The interface classes use this trick to provide more information than a normal signal would provide.

The MonitorPage class contains read-only controls that hold data read back from the Projector Controller board. When a serial port is opened, a signal is fired that contains a Boolean value indicating whether or not the connected board is running application firmware. The MonitorPage class has a slot for that signal called `doDeviceStartAction`. If application firmware is running, this slot starts a timer (`QTimer`) that ticks once per second. On every tick, a signal is fired that is used to read information from the connected board such as fan speed, ADC readings, and so on. The data returned back is converted into human-readable values using the conversion formulas earlier in this document. The data also contains the most recent error from the board, which is printed to the Log.

The SettingsPage class contains some settings the user can configure, such as target temperature, over-temp limit, and the minimum fan speed (duty cycle). This page will refresh its settings when the serial connection is opened if the application firmware is running so that the user sees the most up-to-date board settings. The user can also click the "Refresh" button for the same purpose. Users' modifications to settings are applied when the user edits one of the fields and hits Enter or leaves the field (causing it to lose focus). The settings can be saved to the connected board's built-in EEPROM memory by clicking the "Save" button.

The UpdatePage class allows the user to update the application firmware on the device from an Intel hex file (ends in ".hex"). The user selects a file with the "..." button and then clicks the "Start" button to do the update. The button fires a signal which tells the SerialComm object to perform the update procedure (see the

`doFirmwareUpdate` function in `software/serialcomm.py`). This procdure makes use of the FlashImage class found in `software/flashimage.py` to parse the hex file and hold the firmware application's binary image. The procedure emits signals back to the UpdatePage which increments the progress bar as the update progresses. When complete, a final signal will be emitted to indicate success or failure. Upon selecting a hex file, an MD5 hash is calculated for that file and displayed on the UI. The hex file is read line-by-line as text; no parsing of the hex file is performed for the MD5 hash.

4.3.3 Serial Communication

As previously mentioned, the program entry point in `software/main.py` creates a SerialComm object and then transfers it to its own thread. The SerialComm object does all of the serial communication in repsonse to signals from the UI. It also sends signals back to the UI to indicate the completion and results of commands. The SerialComm class is implemented in `software/serialcomm.py`. Many of the methods in this class are slots for the signals fired by the UI. These are signified by the `QtCore.Slot()` decorator above each method.

The slots have another decorator above them called `_handlesPJCExceptions`, which is implemented in the SerialComm class. This decorator is simply a way to allow all of the slots that communicate over the serial port to handle all of the exceptions the interface code may throw (more on the interface code in a bit) in a standard way. All of the exceptions are defined in `software/pjcexcept.py` and derived from the base PJCError exception also defined in that file. Following are the exceptions defined and what they signify.

NotRespondingError
: The connected device did not respond to a command, meaning it may have lost connection. The serial port will be closed and a message will be printed to the UI Log to explain what happened.

UnknownCommandError
: The device responded with a message saying it does not recognize the sent command. The command may be invalid or intended for the bootloader when the application is running (or vice versa). A message is printed to the UI Log to indicate what happened.

UnexpectedResponseError
: The device responded to a command, but the data was not in the expected format. An example would be getting a hexadecimal value back when the software was expecting a floating-point value. The serial port will be closed and a message will be printed to the UI Log to explain what happened.

DeviceRestartError
: The device restarted at some point since the last command. Functions that purposefully restart the device will need to catch this. A signal will be emitted to the UI indicating device startup containing a Boolean stating whether the device is running the application or bootloader firmware. A message will also be printed to the UI Log.

SerialPortNotOpenError
: The command cannot be sent because the serial port is not open. A message will be printed to the UI Log to explain this.

The actual interface to the device is implemented in `software/pjcinterface.py`. The PJCInterface class defined within is not used directly. It is subclassed by the PJCBootloader and PJCApplication classes in `software/pjcbootloader.py` and `software/pjcapplication.py`, respectively. These subclasses implement commands specific to the bootloader and application firmwares, respectively, and are used in the SerialComm class to actually talk to the device.

The PJCInterface class itself is what actually sends and receives data over the serial port and implements commands common to both the application and bootloader firmwares. Sending a command is done via a single method called `execCommand`, which accepts the command string, a response type, and a timeout period as its arguments. The command string is sent out and the timeout determines how long the method waits for a response. When a response is received, the repsonse type determines how to parse the string into usable data. This data is what is returned. The class defines constants corresponding to the different ways in which the data can be parsed.

Response Type Name	Value	Description
RespString	0	Leave response as string without parsing.
RespDecimal	1	Parse response as decimal integer.
RespHex	2	Parse response as a hexadecimal integer.
RespFloat	3	Parse response as a floating-point number.
RespStatus	4	Parse response as a bootloader status (“!xx”).
RespStrList	5	Response is a list of strings, one per line.
RespDecList	6	Response is a list of decimal integers, one per line.
RespHexList	7	Response is a list of hexadecimal integers, one per line.
RespFltList	8	Response is a list of floating-point numbers, one per line.

The functions that call the commands will specify the appropriate response type. See the PJCBootloader and PJCApplication classes for examples on how these are used.

–End of Document–