Cooling System

ECE 4433: Safety Critical Design Project Report II (Group 2, 2015)

Alexander Ross, Fuad Hussein, Jeffrey Wo

***Abstract*— this report summarizes the various design choices and implementations used to produce a dual-standby safety critical system. Features and test results of the system will be examined in detail. Ways in which the system meets the requirements will heavily reference the Appendices containing various figures and tables located at the end of this report.**

***Index Terms*— Arduino, cooling system, dual-standby, nuclear plant cooling tower prototype design, Peltier Cube, Safety Critical Analysis and Design.**

I. Deliverable One

Introduction

This prototype is a scaled model of a Nuclear Cooling tower accomplished by powering a coil wounded heat generating resistive block. The main objective of the safety system is to achieve a high level of availability by utilizing fail operate procedures that react to mishaps. This is achieved by including two Arduino Due Microcontrollers acting as safety mechanisms. The safety system is programmed with the C based Arduino language which contains range thresholds that flag spurious readings. Communication over serial is used to exchange statuses with the secondary system that is inactive until conditions are met in the primary system. These conditions range from output errors that alert the microcontroller that the effector output is not meeting the required output. This is indicative of an internal issue, or the primary system failing to communicate with the secondary system. If that occurs the lack of communication signals the secondary system to become the active microcontroller.

From the initial design provided in Part 1, the main operation of the whole system did not change. After additional research on the initial choices, key components were replaced with alternatives and additional safety features were added. These changes include; replacing Squareware and mbed with two Arduino Dues, one Peltier cube instead of two, the addition of three relays, 10k pull-up resistors and an emergency switch. These changes are discussed in detail in Section II, Part C.

II. Deliverable Two

A. System Description

The normal operations are defined by powering a heat block while aggressively cooling it with fans and the cooling effect generated by the Peltier cube. Table 1 contains the general components of the prototype with their failure rate per hour and description. The heat sync is the metal dissipating the high temperatures generated by the heat block. To ensure high surface contact between the Peltier cube and the heat block, Arctic Silver 5 thermal paste was applied. Two heat sensors are used at separate locations on the cooling unit. One is dedicated to measuring the heat block, while the other is attached directly to the Peltier cube. To achieve a stronger combined force, two fans are also used. If one fan malfunctions, the system is aware and directs the functioning fan to a higher PWM level. A metal tube surrounds the prototype to guide air upwards. Appendix I contains an image of the prototype.

The hardware diagram and software architecture diagrams that were made to design the system are located in appendices II-IV. Microcontroller A is the primary system communicating serially with Microcontroller B, the standby system. Ways in which the software and hardware mitigate risk is further elaborated upon in subsection B.

Table 1

Part Name and model	**λ (Hour)**	**Function and Description**
Peltier Cube TEC1 12709	1E-6	Cooling the heat block via thermoelectric effect with n-type and p-type semi conductive pellets.
Heat block 1w (Standard)	1E-6	Resistor with internal winding used to generate heat.
Fan AFB0712SHX02	1.65E-6	Motor with blades from a consumer grade heat sync with PWM and RPM signal outputs. Cools metals by directing airflow.
Temperature Sensors TMC50- 25	1E-6	N-type thermoresistor used to measure the temperature of the system.
Relays JQX-15F (787), DC60S5	0.5E-6	Electromagnetic switches that direct power to ground or external PSU. Controlled voltage output to the effectors.
Microcontrollers Freaduino Due MB_DUE_r3	18.9E-6	Controller with CPU and memory. Communicating to detect error and controlling the system.
Terminal boards (Standard)	1E-7	Act as wires to organize and simplify connections.
Wires (Standard)	3E-6	Connect devices and pins.
10k resistors (Standard)	3E-8	Act as pull up resistors for PWM.
Voltage Sensors 10589a	1E-6	Read the voltage passing to the Peltier cube from the power supply
Power Supply (Standard)	1E-6	Supply power to the Fans/Peltier Cube/Heat block

B. Mishap Mitigation Measurements

Following the analysis in the diagram, there are three main sections that have safety implementations to prevent mishaps which are illustrated in appendix XII. The following three methods to mitigate risk are applied in a basic system to minimize probability of a mishap occurring:

1. An external safety measure that is incorporated as a layer of safety between the mishap and hazardous event from the application.
2. An internal safety measure between the hazardous event from the application and individual system failures.
3. The reliability and quality improvement before any system failures.

The internal layer of safety for the cooling system design is a software based detection approach utilizing sensors measured by the microcontrollers to trigger various mitigating techniques which divert the potential hazard [1]. The software functions with the microcontrollers to detect any abnormalities concerning the temperature, tachometer or voltage sensors. When it does it reassigns the associated flag variable and opens the connection to disengage the effectors to prevent the potential mishap. To keep the system fail-operational, different states are defined to take into account individual effector signals and to detach if they cease to operate unexpectedly. An RGB LED is used to reflect the system state. This allows an operator to see when the system is in a critical, normal or standby state. In addition, there is a serial output that can be monitored via a COM port on a computer terminal which updates the operator every second while in the operating state.

The external layer of safety for the system are the relays acting safety interlock. They do this by passing the voltage from the power supply into a common ground when no microcontroller signal is present. This allows the external power to disconnect, minimizing the risk of damaging system components.

Improvement of component reliability and quality was considered. When possible, standard devices were used with a low level of failures per hour. The microcontrollers have a resettable fuse that protects the supplying USB port. This inhibits the board from shorts and overcurrent if the USB power supply was to surge. This also regulates the output for the microcontroller pins to ensuring the components of the microcontroller will not deteriorate or become overloaded. The power supply also has a built in overload protection system and a fixed output to prevent any over powering that would result in component failure.

The mishap mitigation procedures adhere to a list of commands based on five different states identified by the fail detection mechanism. The safety system determines the state (Section II, Part C), by discretely reading values from the temperature sensors, the RPM of the individual fans, the PWM output to the fans (End around), and the voltage passing through the Peltier cube positive terminals (Wrap around). If a component reading exceeds its threshold the system detects the error and flags the variable. Based on the criticality of the component the system will either enter the fail-operate mode or the fail-safe mode. The failure detection mechanism dynamically adjusts the state it is in. If, for example, the conditions to enter fail-operate mode are detected, the safety system continues to perform measurements and adjust the variable flags accordingly. If the flags that were set high are set low, indicating normal operation, the system re-enters the normal state.

C. Safety Feature Design Modifications

The Arduino Squarewear 2 and the mbed were previously specified to be the primary and secondary standby system respectively. Due to pin limitations and different voltage levels having two Arduino Dues in their place was a necessary change. The Arduino Dues have more pins and communicate over Serial without the need of pull resistors, which would have been required with the Squarewear and mbed. This adjustment still maintains the expected fail-operational features of the system with the advantage of a more supple system.

Arduino Due microcontroller were the selected boards built with the AT91SAM3X8E processor [2]. The board has 54 digital I/O pins in which 12 can be used as a PWM output. This is a highly preferred feature of the board since the cooling system needs many I/O pins to control LEDs, fans and relays. It also has 12 analog input pins capable of reading any sensor data to be converted digitally for analysis and serial monitoring. The board is also capable of outputting up to 5 V DC and up to 800 mA. This used for the components in the system that have low power draws. The Arduino Due has powerful interrupt capabilities allowing any pin on the board to operate with the interrupt service routine. See appendix VI and V for pins used and a detailed pin description.

The flexibility available with the Arduino Due is favorable for this cooling design as there are plenty of pins available and each with the capability of becoming an interrupt. This gives the formation of the system wiring for sensors or switches to be easily placed on convenient pins. Both boards also have compatible serial communication links to verify the presence of the other. This makes it possible for the backup system to be notified of a missing link to the primary system when a failure arises.

Due to the limited space on the heat sync, and the Peltier heat dissipation requirements which were previously unknown, only one Peltier cube was incorporated into this system. With no radiator on the Peltier cube, the heat generated outweighs the cooling effect approximately ten seconds after being powered on.

The addition of three relays were introduced into the system. The two fans are similar in voltage and current requirements so they paired to the same relay. The heat block is contained on the same model relay as the fans. The Peltier cube is employed with a solid-state relay rated for higher

current values. The safety system controls whether the signal sends the relays to ground or to the external power supply. This is to ensure minimal component damage and it is an effective way to turn off the power to the heat block in the event of a critical fail. It also allows the system to prep the cooling system prior to the main operations. With this alteration implemented, the system can now continue to operate even when a component connected to one of the relays has failed by simply sending a signal to the components relay from the microcontroller. This feature is necessary and effective for a fail-operational system since failing effectors can be cut off from the main system and also be detected.

Pull-up resistors were required to cleanly read the RPM of the fans. Both of these 10k resistors are being powered by the 5V rail generated from the Arduinos. Because the Arduinos pin capabilities are limited in how much voltage they can sense and the PWM provides a small signal, the pull-up resistors are necessary. A more effective means of achieving this is with a logic level shifter, which was realized too late in the project timeline to be implemented.

From the software scope the system has the ability to detect and flag system failures for effectors to be properly compensated for. The fan speeds can be adjusted via the PWM input signals to a max speed in the case that the Peltier cube has failed and the system must continue to operate. Should all the effectors fail, the system safety features can detect this situation and enter a critical fail-safe state that stops the current flowing to the heat block, ceasing further heat generation. A manual override switch connected to both microcontrollers is programmed to be an external emergency breaker shut down for both systems in the case an unexpected failure or event is occurring. This allows an operator to manually shut down the system if the alarms are issuing warnings, but is unable to repair itself and enter a fail-operate or fail-safe mode.

Dual redundancy was the initial plan within the prototype. Due to the high reliability of sensors, a dual-standby model was adapted instead. This safety feature has been designed to allow the system to keep running in normal conditions if the primary microcontroller has been completely removed or destroyed. This is accomplished by programming a serial link between the two microcontrollers. Signal statuses are constantly sent to the secondary microcontroller from the primary. If there is an absent signal the secondary system assumes the primary system has experienced a failure and begins to operate.

The system architecture flow chart in Appendix II details the various commands that are run based on the system state. The function “Identify State” is implemented by first reading the various sensors and feeding the readings into the range check function that compares the values to a predefined mean value. Included in the range check function is the option to set tolerance. Some components have individual tolerances, or similar components have the same upper and lower limits. Based on the flags, the “Implement State” function sets a flow of different functions, which define five states as detailed below.

1. *Startup:* The microcontroller signals the fan and Peltier relays. The system waits for the Peltier cube/Heat Sync to reach a pre-determined temperature monitored by a sensor mounted on the radiator. Once achieved the state is re-identified.

2. *Normal:* A signal is sent to the heat block relay such that heat is now being generated. The fans and the Peltier cube are both actively cooling in this state. Readings are validated and flagged if an error is detected.

3. *Fail Operate:* This state is entered for a variety of potential mishaps. Where the main task of the system is to cool, more power is delivered to the Peltier cube and the PWM signal to the fans are maxed. For the system to enter this state a fan or the Peltier cube may be flagged, provided the other two fan flags are low.

4. *Fail Safe:* This state is outside of the scope of the main operation. It is the state the system is in while shutting down to mitigate a potential hazard. This is triggered if both fans and the Peltier cube are flagged. Once entered the system must be reset.

5. *Standby:* The system enters a standby state where nothing should be executed and all effectors should be disconnected allowing the system to be in a fail-safe non-operable state. [1]

III. Deliverable Three

A. *F.M.E.A.*

Failure mode and effect analysis (FMEA) was performed on the system to identify the various components in the prototype and the effect of each failure within the system. It is effective for identifying hazards in order to implement fail operate techniques. See Appendix XI for the full page report.

B. *F.T.A.*

For the full system analysis Failure Tree Analysis (FTA) can be found in appendix VIII-XI. The system has one critical error, this being that the heat block reaches high temperatures without sufficient cooling to dissipate the heat. The FTA shows how the prototype employs the relays as a safety interlock mechanism. As discussed in previous sections, with the relays controlling the route the power supply travels, a critical failure could only occur by switching mechanisms in the relays. The probability of that happening is 0.5E-06, which is within reasonable standards.

[1] Note: Microcontroller B is defaulted to this state. Once left, A does not assume standby roll. Microcontroller failure post state change results in a fail safe state.

C. R.A.

To verify that the risk mitigation techniques that were used lead to an acceptable level of mishap risk, Risk Analysis (RA) was used. The various values that define the probability values were extracted from [3] and shown in tables 1 and 2. Equation 1 is a high level branch. Appendix X contains the full calculations used to derive the probability of failures per hour. Table 2 contains the actual uncertainty values and failure rates per hour in more detail, excluding the values provided in Table 1 located in the Section II.

Table 2

Module	**λ (Hour)**	**Uncertainty Factor**
Processor	18.9E-06	10
Memory	13.0E-06	10
Relay output	9.2E-06	10
A/D converter	10.4E-06	10
Analog input	15.5E-06	10
Communication (Bus controller)	19.8E-06	10
Electric power supply	33.0E-06	10
Discrete output	1.65 E-06	10

$$P_{OH} = P_{SF} + P_{BF} + P_{PF} + P_{RF} \quad (1)$$
$$= 1.02 * 10^{-5}/Hour$$

D. FMET

Table 3

Failure Type	**Effect**	**Pass/Fail (Safety system response)**
1. Heat generated exceeds 70 degrees Celsius	Heat sensors flagged, fail safe entered	Pass. The system successfully detected the high readings. After fail safe the system must be rebooted.
2. Peltier Cube loses power or experiences power surge.	System enters fail-operate mode, PWM to fan increases. System remains stable.	Pass With both fans being wrote a higher PWM value the hot air quickly becomes dissipated.
3. Power to Microcontroller A/B is removed	A/B reacts and continues previous state operations.	Pass Serial listening in Standby is triggered to exit and enter the Normal mode to re-identify the state. If B goes down (Secondary) signal to relays is lost and heat dissipates
4. Full power loss to relays	No power to heat relay, heat dissipates with time.	Pass The voltage being passed from the power supply goes to ground due to the electromagnetic switching mechanism inside the relay.

Failure Mode and Effect Testing (FMET) was limited given the nature of the failsafe mechanism. Event tree analysis was used to understand the failures, located in appendix VII. The safety system mechanisms covered a variety of possible failures to the extent that injecting the failures was limited. Table 2 Operational Hazard Analysis methods was used to prevent the system from failing. By detecting the path that leads to a hazard which causes the mishap the full effect of the hazard can be better understood. For an example; the temperature sensor can either read higher or lower than the actual temperature. This will cause the microcontroller to not trigger or disconnect the effectors correctly which can cause the cooling system to overheat or issue faulty commands. The fan will fail to operate at correct speeds if the PWM readings are corrupted, while the Peltier cube will fail if the temperature sensor is outside of temperature range. Providing too much power to the system could result in burning out or damaging any of the effectors or the micro-controller. In any of these cases the system will go into fail-operate mode before safely shutting down or continue to operate until the main system fails. The environment can also cause the system to fail during a natural disaster or strong weather conditions.

Another source of failure for the safety system can be induced by the mechanical failure in the relays. If the relays fails to disconnect there will be over-cooling or overheating in the system resulting in system failures. The power supply for the units could also fail unexpectedly by electrical failure internally within the supply or improper maintenance in the unit. This failure could cause a ripple effect by causing pin breakout in the microcontroller or in a worst case scenario if the resettable polyfuse within the microcontroller is burnt out it is very likely the board can breakout from overcurrent.

IV. Deliverable Four

A. System Critical Failure Rate Calculations

These equations are used to identify the probability of the software failing. Equation 1 will be substituted to calculate the software critical failure rate further on in calculations. The completed code can be found at appendix XV.

$$N_{OP} = L * K_{OP} \quad (2)$$

N_{OP}: Number of faults in the operational system.
K_{OP}: Software Density. Ranges from **1.1E-2** to **4.4E-4**. Nominal = **2.2E-3.**
L: Line of codes. 597 lines.

N_{OP} yields to be 2.2E-03 * 597 = **1.31** fault.

After calculatingN_{OP}. It is possible now to find the software critical failure rate by using equation 2.

$$\lambda_{Crit} = \frac{N_{Crit}}{T} \quad (3)$$

λ_{Crit}: Average software critical failure rate.
N_{Crit}: Number of fault that can cause critical fail.
T: Total time software is operating.

N_{Crit} must be calculated first in order to find λ_{Crit}. This can be achieved by using equation 3.

$$N_{Crit} = N_{Op} * F_{Crit} \quad (4)$$

F_{Crit}: Number of critical faults occurring from total faults. This was obtained by testing how many times a fault occurs that makes the system fail. This value is assumed to be **1.42E-1** as it is more applicable. N_{OP} can be calculated by using equation 2.

$$N_{Crit} = 1.13 * 1.42 * 10^{-1} = 16 * 10\text{^} - 2 \quad (5)$$

The average software critical failure rate can be obtained now by substituting the obtained the value into equation 2. Furthermore assuming the software is operating for an entire year, which is equivalent to 8766 hours.

$$\lambda_{Crit} = 16 * \frac{10^{-2}}{8766} = 1.83 * 10^{-5} \quad (6)$$

The average software critical failure rate is in safe range. This is comparative of the failure rate of a nuclear power (1e-5).

B. Conclusion

The safety system has met expectations. The normal operation of the system prevents the heat block from ever reaching the predefined threshold. As previously mentioned the maximum observed temperature of the heat block surpassed the operating range of the temperature sensors. With the maximum output possible with the power supplies the system never surpasses the pre-defined thresholds. With the dual standby measurements being employed one of the microcontrollers can completely fail and the prototype will still have a fully functional cooling system. Due to the external power disconnect provided by the relays, complete power loss to the rest of the system could occur and power being drawn from the high voltage components is diverted to ground. A high level of reliability and availability was achieved with this design.

V. References

[1] Dr. E. C. Guerra, " ECE 4433," UNB, Fredericton, 2015.

[2] Elecfreaks, 2015. [Online]. Available: http://www.elecfreaks.com/store/freaduino-due-mbdue-p-520.html.

[3] A. R. H. P. E. F. Riccardo Manzini, "Maintenance for Industrial Systems," in *2011*.

VI. Acknowledgements

Gratitude is given to Dr. Eduardo Castillo Guerra for his guidance in part selection and design techniques.

VII. APPENDIX

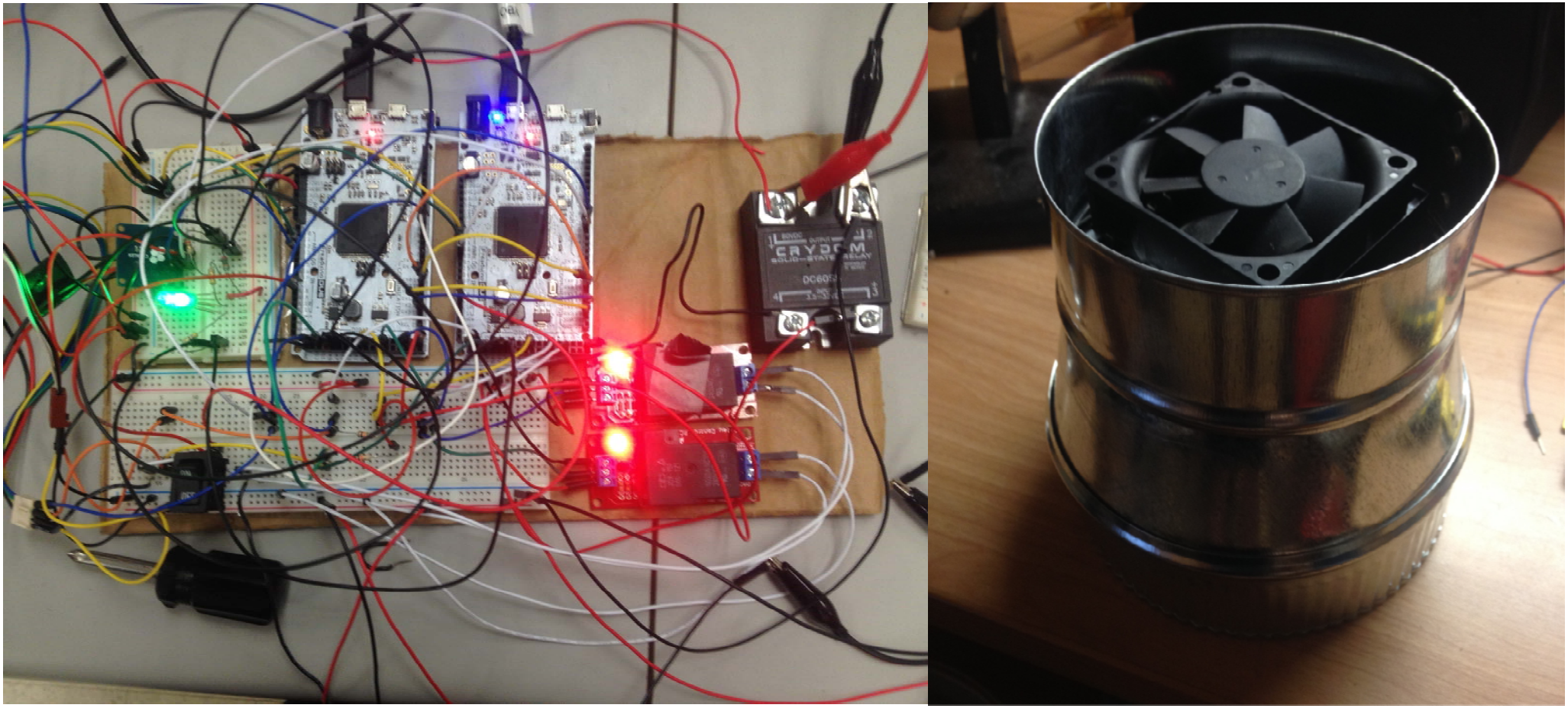

Figure 1: Actual prototype

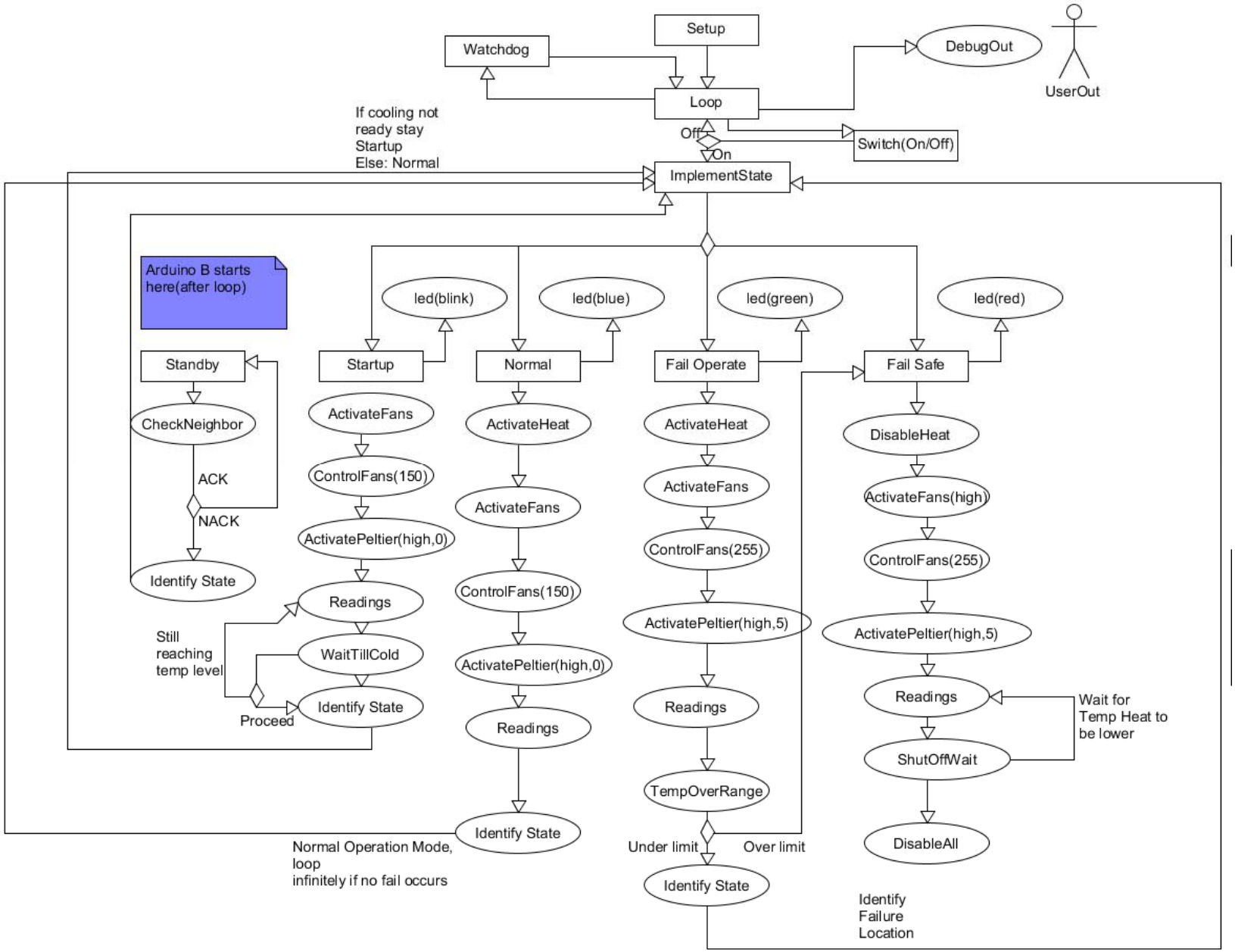

Figure 2: System Architecture (AR)

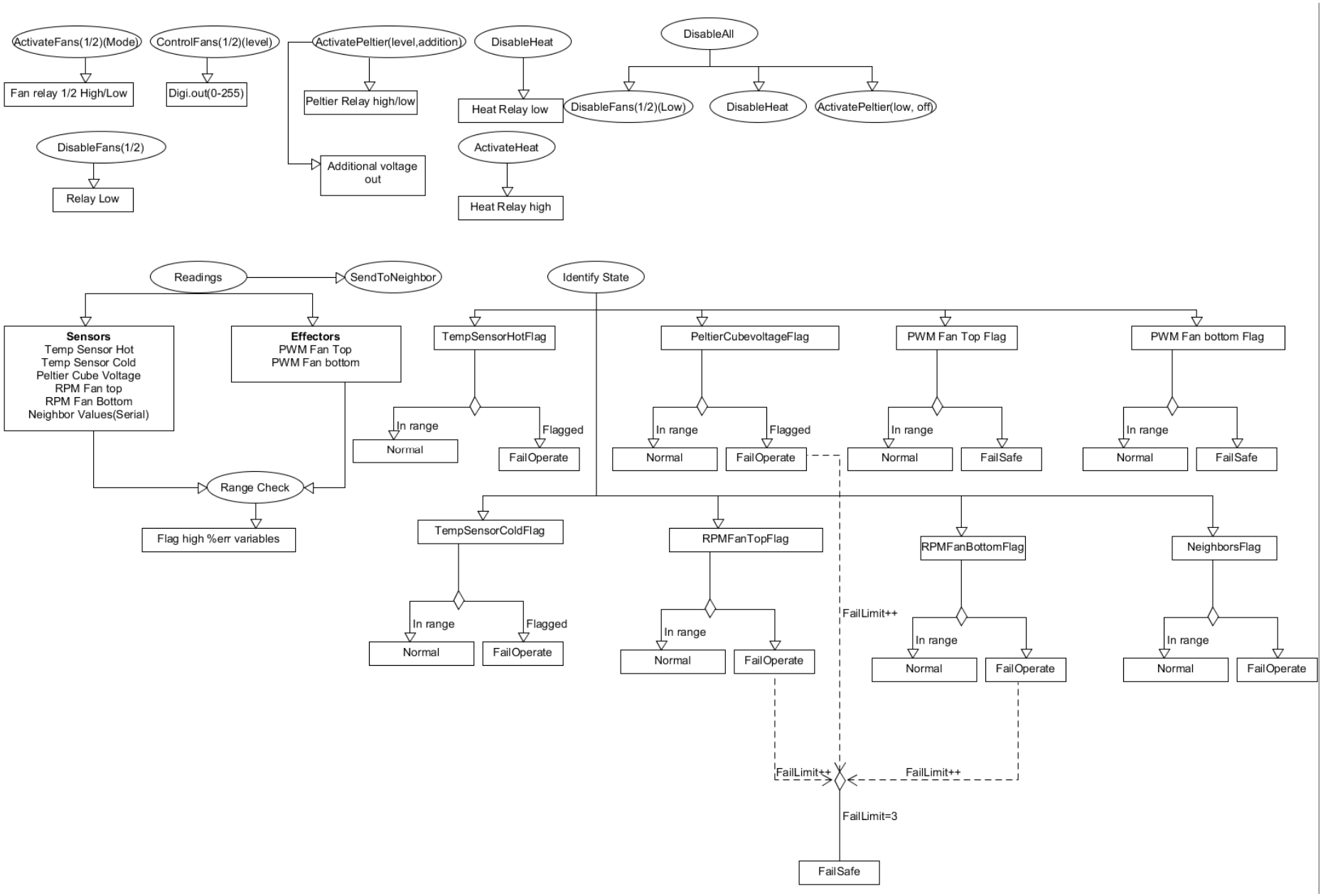

Figure 3: System architecture functions (AR)

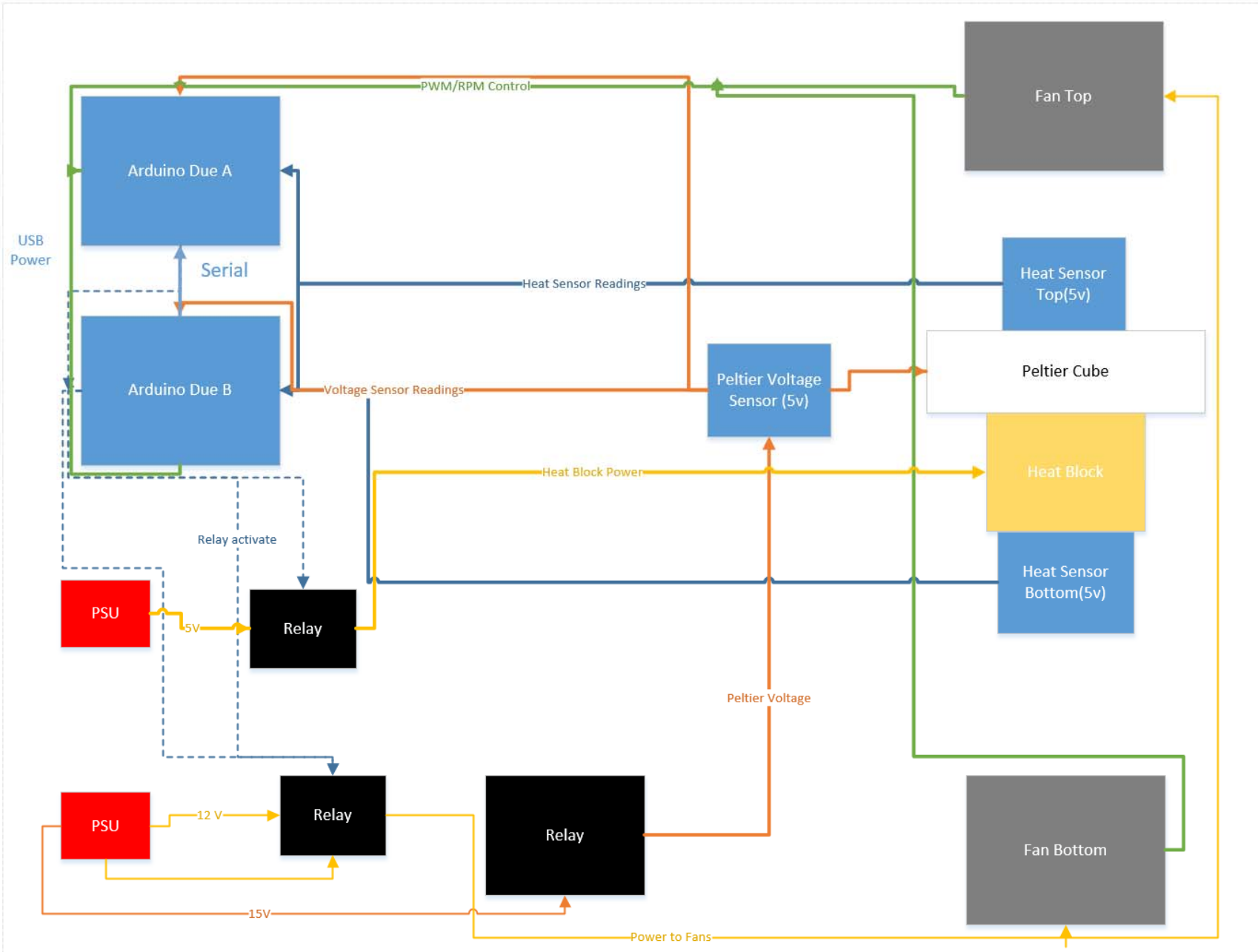

Figure 4: Block Diagram (AR)

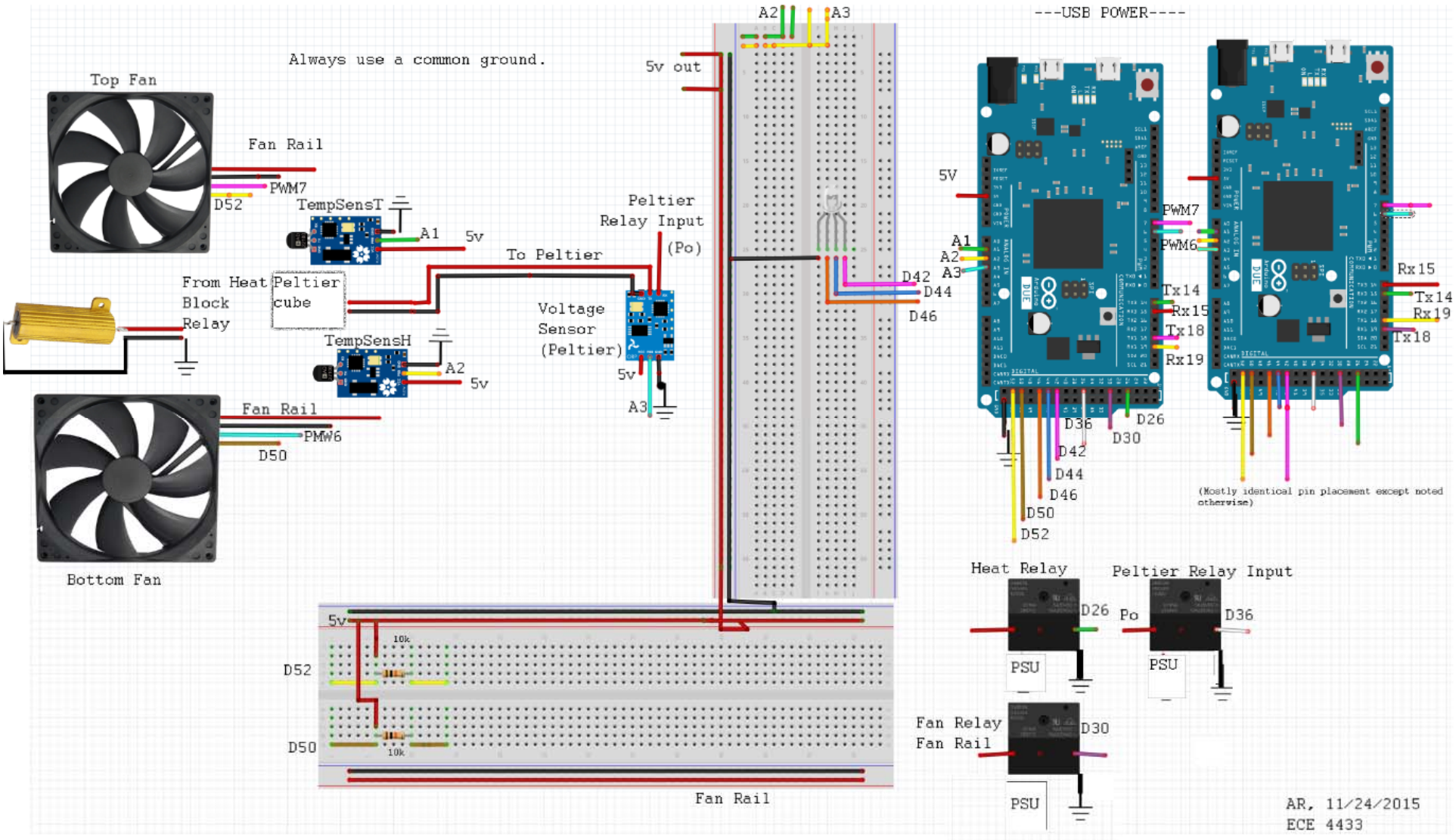

Figure 5: Hardware implementation

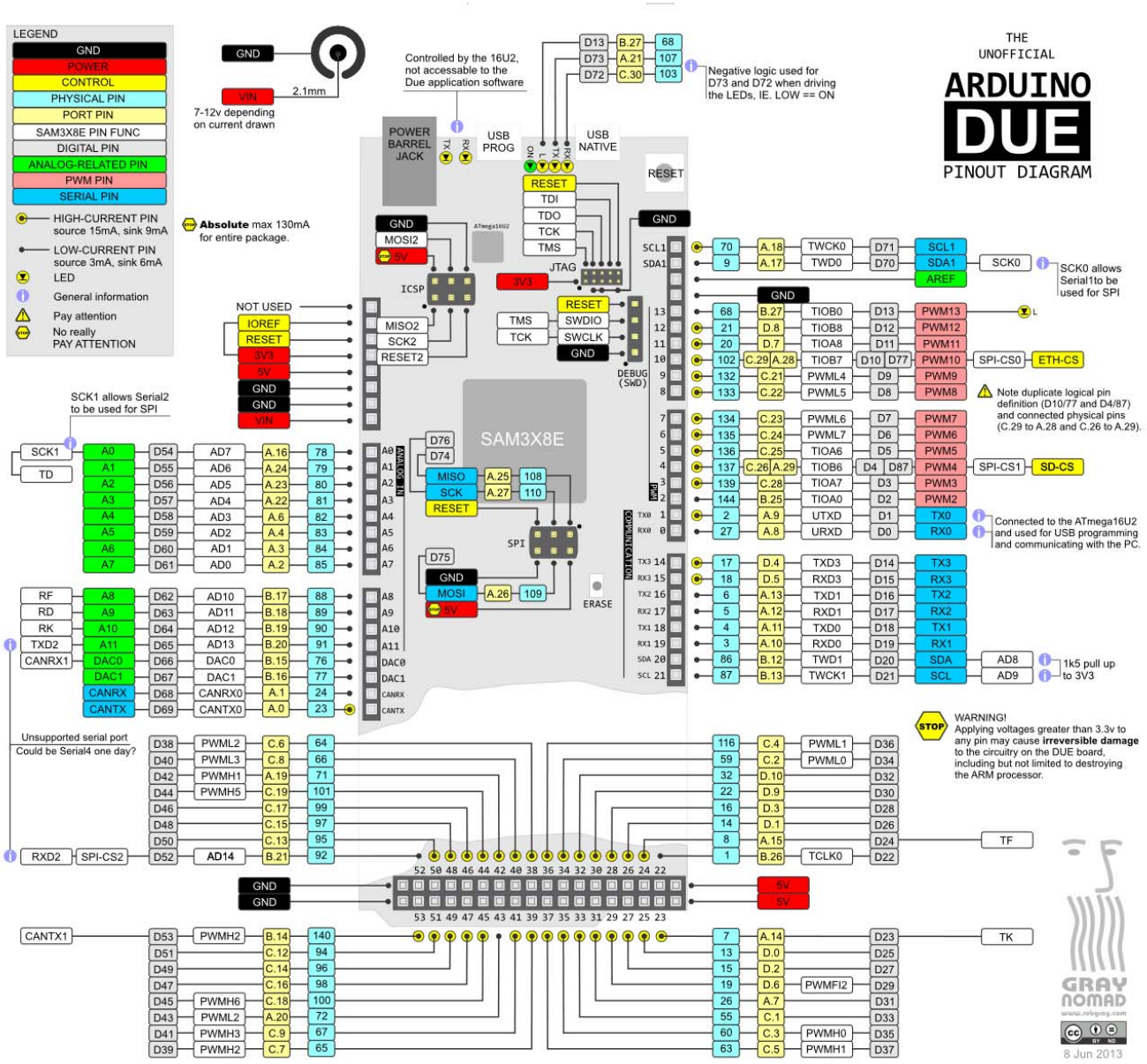

Figure 6: Arduino Due pinout

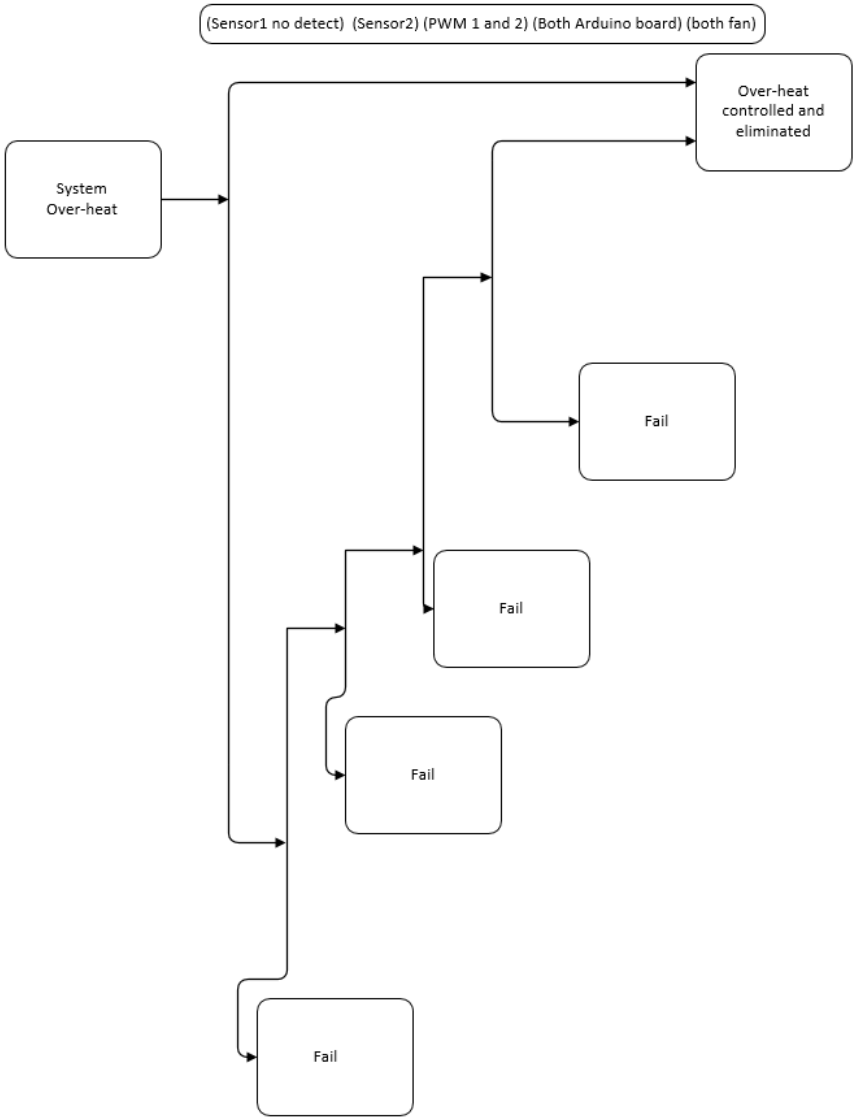

Figure 7: ETA (FH)

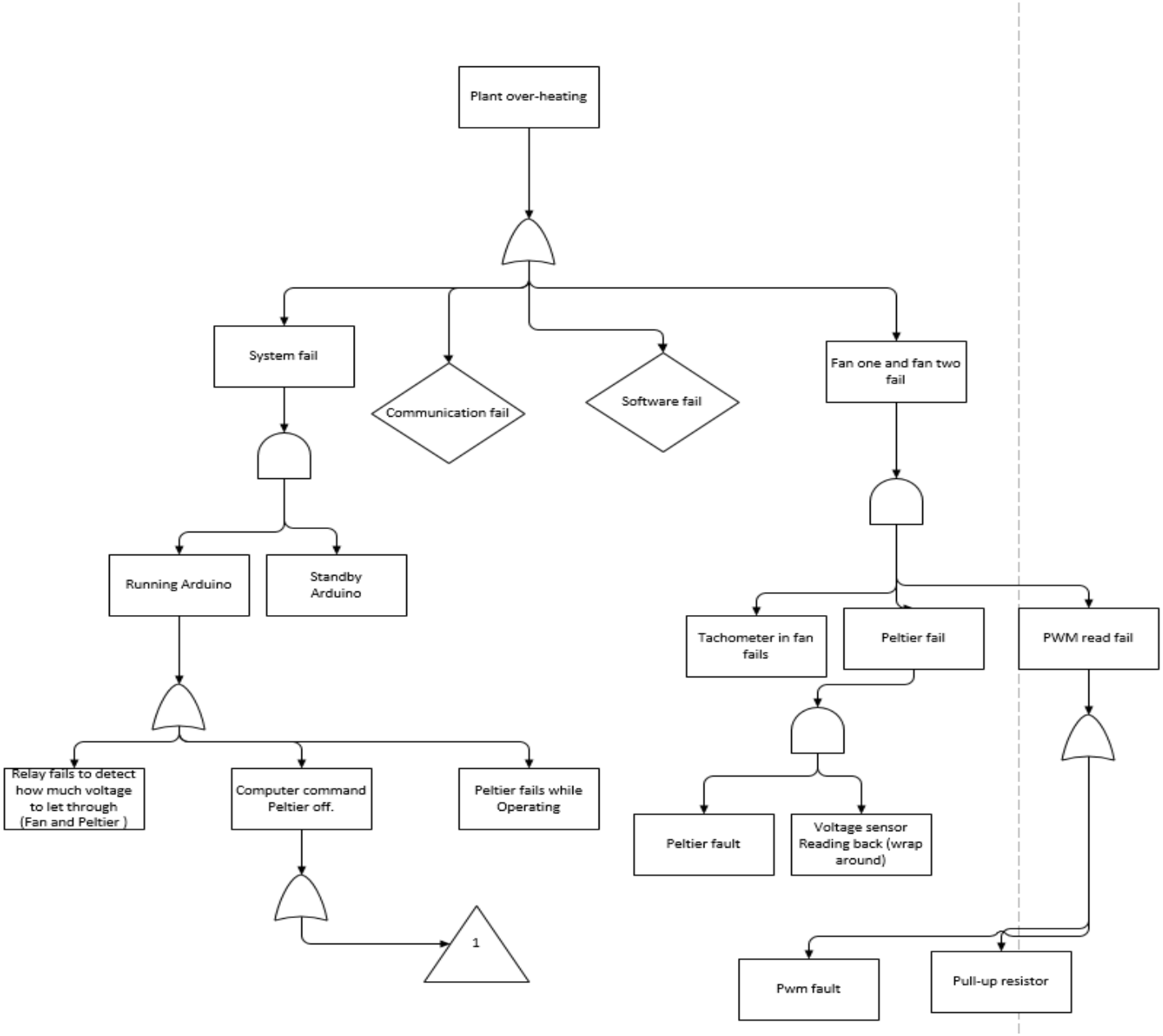

Figure 8: FTA part1 (FH)

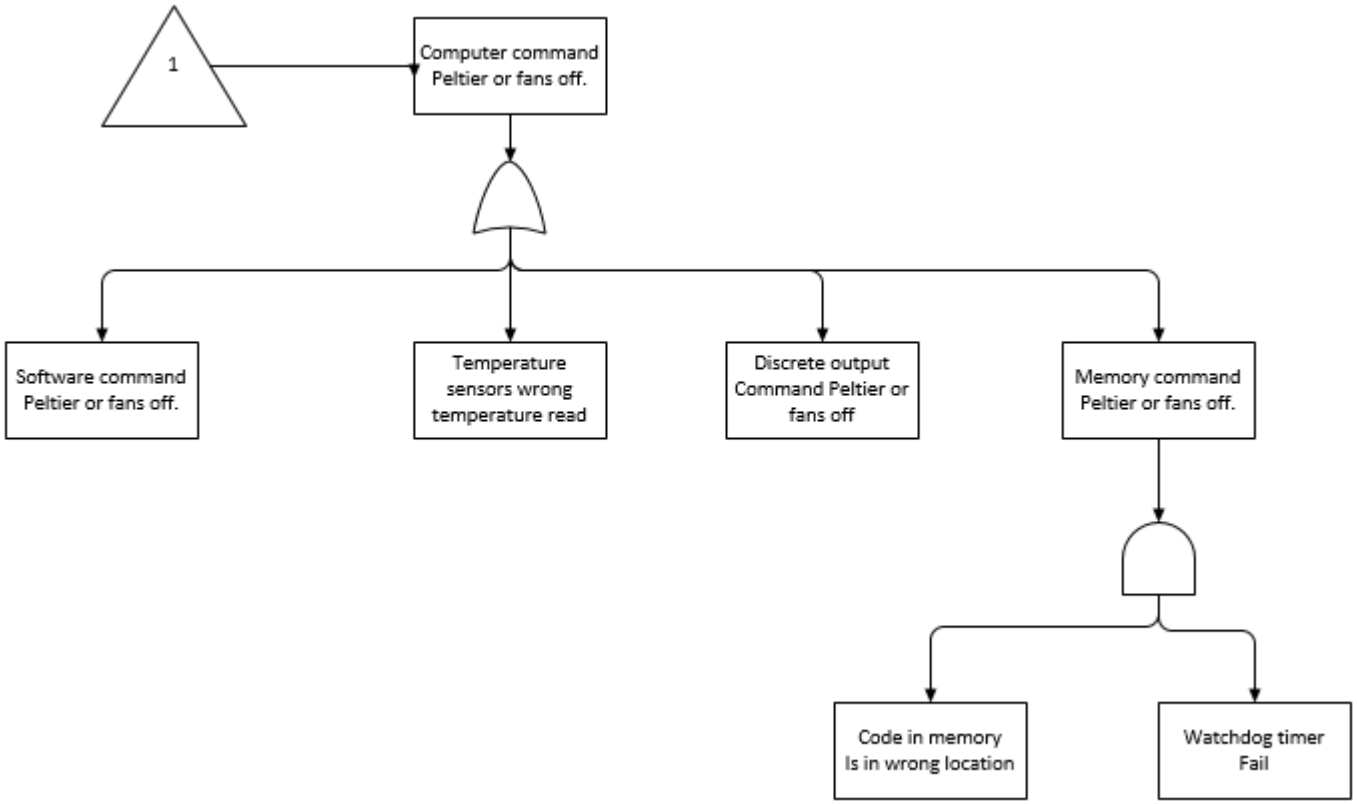

Figure 9: FTA part2 (FH)

$$P_{OH} = P_{SF} + P_{CF} + P_{SF} + P_{F12}$$
$$P_{f12} = P_{FST} * P_{Pf} * P_{PWM}$$
$$P_{pwm} = P_{pwmf} + P_{RF}$$

$$P_{pf} = P_{pfault} * P_{Vs} = 2E - 06$$

$P_{SF} = P_{AB_1} * P_{AB_2}$
$P_{AB_1} = P_{RF} + P_{CFP} + P_{PF}$
$P_{CFP} = P_{SH} + P_{SENSOR} + P_{DO} + P_{MF}$
$P_{MF} = P_{PWM} + P_{WDT}$
$Final\ Calc$:
$P_{MF} = 1 * 10^{-2} * 13 * 10^{-6} = 1.3 * 10^{-7}$
$\mathrm{P_{CFP}} = 1.83 * 10^{-5} + 1 * 10^{-6} + 1.15 * 10^{-5} + 1.3 * 10^{-7} = 3.6 * 10^{-5}$
$\mathrm{P_{AB_1}} = 9.2 * 10^{-6} + 3.6 * 10^{-5} + 1 * 10^{-6} = 4.62 * 10^{-5}$
$\mathrm{P_{AB_1}} = \mathrm{P_{AB_2}} \rightarrow \mathrm{P_{SF}} = (4.62 * 10^{-5})(4.62 * 10^{-5}) = 2.13 * 10^{-9}$
$P_{pf} = (1 * 10^{-6})^2 = 1 * 10^{-12}$
$P_{pwm} = 1 * 10^{-6} + 1 * 10^{-6} = 2 * 10^{-6}$
$P_{f12} = 1 * 10^{-6} * 2 * 10^{-6} * 2 * 10^{-24} = 2 * 10^{-24}$

$$P_{OH} = 2.13 * 10^{-9} + 1.83*10\text{^-}5 + 19.8*10\text{^-}6 + 2*10\text{^-}24$$

$$P_{OH} = 3.8 * 10^{-5}\ /Hour$$

Figure 10: Full Risk Analysis Calculations (FH)

Fail Mode and Effect Analysis (FMEA)

System : Cooling fan system page : 1 of 1

Subsystem: All Date : Nov25 / 2015

Operation mode: Operating
Fuad Hussein

Component	Failure Mode	Failure Effect
-Temperature sensor	-Reads high -Reads low	-Fan does not operate according to sensor reading, fan speed will be higher or lower than what is required.
-Speed sensor	-Saturation -Burn out (resistor & Amp)	-The feedback read will disconnect once it detects inaccurate PWM signal.
-Fan	-Not Operating -Wrong speed for temperature	-The cooling will not be new the required temperature (PWM error)
-Power	-Transient -Off -Wrong power	-A turn off or burn for the fan and micro-controller
-ADC	-Error in sampling -Burn out	-An error in micro-controller decisions due to wrong readings.
-PWM	-Stuck (on/off)	-Signal controller -Power to the fan will assume full output.
-Peltier Cube	-Unresponsive -Burn out -Error in voltage driving value	-Other Peltier cube will compensate and enter safe shut down -Fan will compensate
-Relay	-Fails to control voltage.	-The fan will burn once the relay fail to limit the voltage sent to it. -The Peltier cubes will also fail due to voltage overload.
-Arduino board	-Fail to communicate and detect error.	-The system will fail since the backup did not detect the fail.
-wire connections	- Open or shorted.	-The circuit will not connect the system as required.
-LEDS	-Stuck on or off	-Error in mode display
-10k resistor	- Not working as a pull up resistor.	-Cannot read the value of the RPM.
-Voltage sensor	-Error in reading voltage value	-Error in reading the voltage value of the Peltier cube
-Switch	-Stuck on and off	-Error on standby mode display.

Figure 11:FMEA (FH)

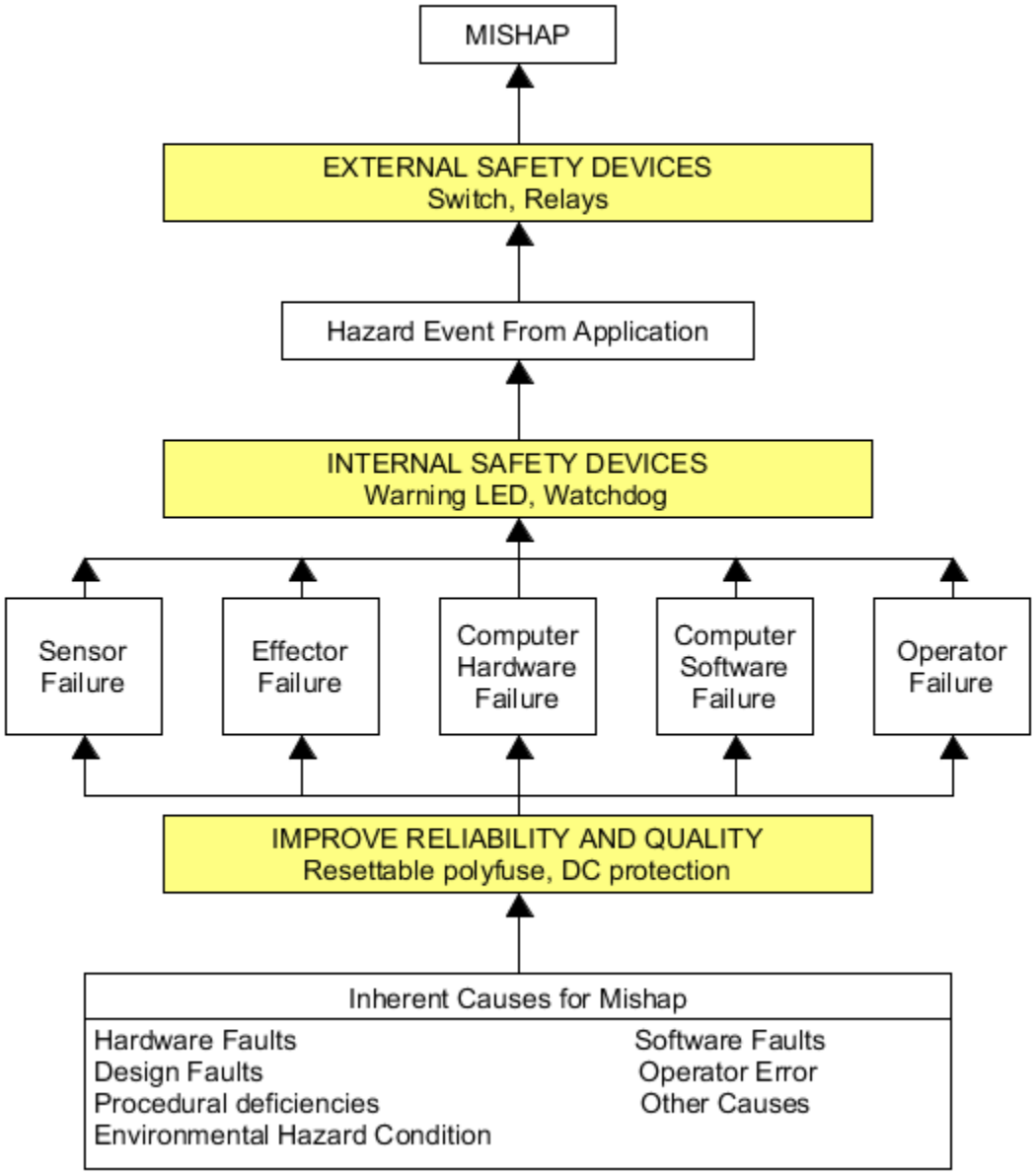

Figure 12: Mishap mitigation risk techniques (JW)

Code

```
#include<math.h>

//To do: Setup, output readings, identify state


/*pin quick look up

*Temp
*A1 is sensor A
*A2 is sensor B

*Fan RPM
*A26 is top fan RPM reading
*A27 is bottom fan ROM reading

Peltier voltage sensor on A5
*Relays
* fan d22
* heatblock d24
* peltier A8

Bottom PWM on PWM 3
Top PWM on PWM 4

-Serial: 0 (RX) and 1 (TX)
-Serial1: 19 (RX) and 18 (TX)
-Serial2: 17 (RX) and 16 (TX)
-Serial3: 15 (RX) and 14 (TX)


*/

//Switch
int switchpin = A8;

int peltier_pin=A3;

//Sensor 1 Values
int tempPinC = A1;
int vdrduinoC= 5;
int tempflagC = 0;
double conv_Cold;
double conv_Hot;
double conv_Pelt;

//Sensor 2 Values
int tempPinH = A2;
int vdrduinoH = 5;
int tempflagH = 0;
float tempH=0.0;
float tempValueH=0.0;
int val2;                //Create an integer variable

unsigned char c;



//RPM values
unsigned long lastmillis = 0;//Interrupt for Hall
Effector (RPM)

/*Top Fan */
volatile int rpmcount_t = 0;
int rpm_t =0;
int rpm_dpin_t=52;//Digital pin 26
int topFanPin= 7;//Analog output pwm


volatile int rpmcount_b = 0;
int rpm_b =0;
int rpm_dpin_b=50; //Digital pin 27
int bottomFanPin=6; //Analog output pwm


/*Relay pins*/

int relay_fans = 30;
int relay_heatblock = 26;
int relay_peltier = 36;


/* Overall State of System Identifiers*
 * 0 startup, 1 normal, 2 fail operate, 3 failsafe, 4
Standby*/
 int State=0;
```

```
 /* Neighbor State of System Identifiers*
 * 0 startup, 1 normal, 2 fail operate, 3 failsafe, 4
Standby*/
int neighbor_state=0;

//Flags
int TempSensorHotFlag=0;
int TempSensorColdFlag=0;
int PeltierCubeVoltageFlag=0;
int RPMFantopFlag=0;
int RPMFanBottomFlag=0;
int NeighborValuesFlag=0;
int PWMFanTopFlag=0;
int PWMFanbottomFlag=0;

//measurement readings
int raw_TempSensorHot=0;
int raw_TempSensorCold=0;
double raw_PeltierCubeVoltage=0;
int raw_RPMFantop=0;
int raw_RPMFanBottom=0;
int raw_NeighborValues=0;
int raw_PWMFanTop=0;
int raw_PWMFanbottom=0;

/*Predefined reading values for range check THIS IS
NEEDED!!!*/

int tolerance= 10000;
double pre_TempSensorHot=50;
double pre_TempSensorCold=0;
double pre_PeltierCubeVoltage=0;
int pre_RPMFantop=0;
int pre_RPMFanBottom=0;
int pre_NeighborValues=0;
int pre_PWMFanTop=0;
int pre_PWMFanbottom=0;

char reading_list_string;
char compared_string;
char neighborReadings;

// Internal connections LED:
// - red:   D42
// - green: D44
// - blue:  D46


#define red   42
#define green 44
#define blue  46

char str[4];

void setup() { //This needs work!

  pinMode(relay_fans, OUTPUT);
  digitalWrite(relay_fans, LOW); // Default top fan to
Off State

  pinMode(relay_heatblock, OUTPUT);
  digitalWrite(relay_heatblock, LOW); // Default block
fan to Off State

  pinMode(relay_peltier, OUTPUT);
  analogWrite(relay_peltier, 0); // Default Peltier to
Off state

  pinMode(tempPinC,INPUT);   // sets the pin as input
  pinMode(tempPinH,INPUT);   // sets the pin as input

  pinMode(peltier_pin, INPUT);

  Serial.begin(9600);
  Serial1.begin(9600);
  Serial2.begin(9600);

  pinMode(rpm_dpin_t,OUTPUT);
  pinMode(rpm_dpin_b,OUTPUT);

  pinMode(red, OUTPUT);
  pinMode(green, OUTPUT);
  pinMode(blue, OUTPUT);
  digitalWrite(red, HIGH);
  digitalWrite(green, HIGH);
  digitalWrite(blue, HIGH);
}

void FansOff(){
  digitalWrite(relay_fans, LOW);
```

```
}

void FansOn(){
  digitalWrite(relay_fans, HIGH);
}

void HeatblockOff(){
  digitalWrite(relay_heatblock, LOW);
}

void HeatblockOn(){
  digitalWrite(relay_heatblock, HIGH);
}

void ActivatePeltier(int state, int additional){

  analogWrite(relay_peltier, state);
}

float readTempSensorsC(){
  // get values from the sensors
  raw_TempSensorCold = analogRead(tempPinC);   //Reading
Through A0 Temperature from the Sensor
  return(raw_TempSensorCold);
}

float readTempSensorsH(){

    raw_TempSensorHot = analogRead(tempPinH);
//Reading Through A0 Temperature from the Sensor
    return(raw_TempSensorHot);
}

void debugDisplay(){
  String out;// = "arduino";
  if(State==3){
    out= "In fail safe mode";
    }
  if(State==2){
    out= "In fail operate mode";
    }
  if(State==1){
    out= "In normal mode";
    }
  if(State==0){
    out="Start up mode";
  }
    if(State==4){
    out="Standby mode";
}
  Serial.println(out);
  Serial.println("Local
Readings_____
_____
__________");
  Serial.println("| Temp Sensor Cold | Temp Sensor Hot |
Converted Peltier | RPM Fan Bottom | RPM Fan top | PWM
Fan Top | PWM Fan bottom | Neighbor Readings |");
  Serial.print("|   ");
  Serial.print(conv_Cold);
  Serial.print("   |   ");
  Serial.print(conv_Hot);
  Serial.print("   |   ");
  Serial.print(conv_Pelt);
  Serial.print("   |   ");
  Serial.print(raw_RPMFanBottom);
  Serial.print("   |   ");
  Serial.print(raw_RPMFantop);
  Serial.print("   |   ");
  Serial.print(raw_PWMFanTop);
  Serial.print("   |   ");
  Serial.print(raw_PWMFanbottom);
  Serial.print("   |   ");
  Serial.print(neighborReadings);
  Serial.print("   |");


}
void fanRPM_t(){
 attachInterrupt(digitalPinToInterrupt(rpm_dpin_t),
rpm_fan_t, FALLING);//interrupt zero (0) is on pin
two(2).
 if (millis() - lastmillis == 1000){  /*Uptade every one
second, this will be equal to reading frecuency (Hz).*/

 detachInterrupt(0);    //Disable interrupt when
calculating

 rpmcount_t = 0; // Restart the RPM counter
 rpm_t=0;
```

```
 lastmillis = millis(); // Uptade lasmillis
  attachInterrupt(0, rpm_fan_t, FALLING); //enable
interrupt

  }
}

void rpm_fan_t(){ /* this code will be executed every
time the interrupt 0 (pin2) gets low.*/
    rpmcount_t++;
}

int fanRPM_b(){
 attachInterrupt(digitalPinToInterrupt(rpm_dpin_b),
rpm_fan_b, FALLING);//interrupt zero (0) is on pin
two(2).
 if (millis() - lastmillis == 1000){  /*Update every one
second, this will be equal to reading frecuency (Hz).*/

 detachInterrupt(0);    //Disable interrupt when
calculating

 rpmcount_b = 0; // Restart the RPM counter
 rpm_b=0;
 lastmillis = millis(); // Uptade lasmillis
 attachInterrupt(0, rpm_fan_b, FALLING); //enable
interrupt
 return(0);
  }
}

void rpm_fan_b(){ /* this code will be executed every
time the interrupt 0 (pin2) gets low.*/
    rpmcount_b++;
   }

double tempConverter(int RawADC) {  //Function to
perform the fancy math of the Steinhart-Hart equation
  double Temp;
  RawADC = map(RawADC,0,1023,1023,0);
  Temp = log(((10240000/RawADC) - 10000));
  Temp = 1 / (0.001129148 + (0.000234125 +
(0.0000000876741 * Temp * Temp ))* Temp );
  Temp = Temp - 283.15;              // Convert Kelvin
to Celsius
  return Temp;
}

void controlPWMFanTop(int pwmod){

      analogWrite(topFanPin, pwmod);  // analogRead
values go from 0 to 1023, analogWrite values from 0 to
255

}

void controlPWMFanBottom(int pwmod){
  analogWrite(bottomFanPin, pwmod);  // analogRead
values go from 0 to 1023, analogWrite values from 0 to
255
}

double peltierReading(){
  double volt;
  volt=analogRead(peltier_pin);
  (5/1023)*volt;
  return(volt);
}

void implementState(int State){
  if(State==0){
    Startup();
  }
  if(State==1){
    Normal();
  }
  if(State==2){
    FailOperate();
  }
  if(State==3){
    FailSafe();
   }
  if(State==4){
    Standby();
  }
  else{
    FailSafe();
  }
}
```

```
void neighborState(int i){

  if(i==5){
    FailSafe();
  }
  else{
    neighbor_state=i;
}

}

void coolingDelay(){
  while(raw_TempSensorCold>15){ //THIS WILL NEED TO
CHANGE!!!!!!! 100 is guessed raw temperature. Need ADC
value.
    raw_TempSensorHot=readTempSensorsH();      //Read
the analog port a2 and store the value in val
  }

}

void Startup(){
  HeatblockOff();
  FansOn();
  ActivatePeltier(255,0);
  controlPWMFanTop(100);
  controlPWMFanBottom(100);
  coolingDelay();
  readings();
  identifyState();

  digitalWrite(green, LOW);
  digitalWrite(red, HIGH);
  digitalWrite(blue, LOW);


}

void Normal(){
  HeatblockOn();
  FansOn();
  ActivatePeltier(255,0);
  controlPWMFanTop(100);
  controlPWMFanBottom(100);
  readings();

  digitalWrite(green, LOW);
  digitalWrite(red, HIGH);
  digitalWrite(blue, HIGH);
  identifyState();


}
void tempOverRange(){
   raw_TempSensorCold=readTempSensorsC();
   raw_TempSensorHot=readTempSensorsH();
  if(raw_TempSensorCold>80){
    State = 3;  //Fail Safe, Bring system to safe (NON
operating) condition
   }
   if(raw_TempSensorHot>80){
    State = 3;  //Fail Safe, Bring system to safe (NON
operating) condition
   }
}

void FailOperate(){
  HeatblockOn();
  FansOn();
  ActivatePeltier(255,255);
  controlPWMFanTop(255);
  controlPWMFanBottom(255);
  readings();
  tempOverRange();

  digitalWrite(green, HIGH);
  digitalWrite(red, HIGH);
  digitalWrite(blue, LOW);
  identifyState();

}

void FailSafe(){
  HeatblockOff();
  FansOn();
  ActivatePeltier(255,255);
  controlPWMFanTop(255);
  controlPWMFanBottom(255);

  digitalWrite(green, HIGH);
```

```
  digitalWrite(red, LOW);
  digitalWrite(blue, HIGH);
  DisableAll();


}

void DisableAll(){
  shutOffWait();
  ActivatePeltier(0,0);
  FansOff();
  readings();
}

void Standby(){//slaves starts here
  int i=0;
   do{
  if (Serial1.available()) {
    delay(500); //allows all serial sent to be received
together
    while(Serial1.available()) {
      str[i++] = Serial1.read();
    }
      a=str[i++];
      Serial.print(a);
  }

  if(i>0) {
    Serial.print(str);
  }
  } while(a==b);

  Serial.println("No signal");
  State=Normal;
}
void shutOffWait(){

   while(raw_TempSensorHot>30){ //THIS WILL NEED TO
CHANGE!!!!!!! 100 is guessed raw temperature. Need ADC
value.
    raw_TempSensorHot=readTempSensorsH();      //Read
the analog port a2 and store the value in val
  }
}
```

```
void watchDog(){

  const int RSTC_KEY = 0xA5;
  RSTC->RSTC_CR = RSTC_CR_KEY(RSTC_KEY) |
RSTC_CR_PROCRST | RSTC_CR_PERRST;
  while (true);

}
int readings(){

  int reading_list[100]={0, 0, 0, 0, 0, 0, 0};

   raw_TempSensorCold=readTempSensorsC();      //Read
the analog port a1 and store the value in val
   conv_Cold=tempConverter(raw_TempSensorCold);

TempSensorColdFlag=rangeCheck(conv_Cold,pre_TempSensorCo
ld,tolerance);

   raw_TempSensorHot=readTempSensorsH();      //Read the
analog port a2 and store the value in val
   conv_Hot=tempConverter(raw_TempSensorHot);

TempSensorHotFlag=rangeCheck(conv_Hot,pre_TempSensorHot,
tolerance);


   raw_RPMFantop= rpmcount_t;       //top fan rpm
   RPMFantopFlag =rangeCheck(raw_RPMFantop,
pre_RPMFantop, tolerance);//flag

   raw_RPMFanBottom = rpmcount_b;   //bottom fan rpm
   RPMFanBottomFlag=rangeCheck(raw_RPMFanBottom,
pre_RPMFanBottom, tolerance);//flag

   conv_Pelt=peltierReading(); //voltage through peltier
voltage sensor

PeltierCubeVoltageFlag=rangeCheck(conv_Pelt,pre_PeltierC
ubeVoltage,tolerance);//flag


   raw_PWMFanTop=0;//end around test for pwm on fan top
   PWMFanTopFlag=rangeCheck(raw_PWMFanTop,pre_PWMFanTop,
tolerance);//flag
```

```
   raw_PWMFanbottom=0;//end around test for pwm on fan
top

PWMFanbottomFlag=rangeCheck(raw_PWMFanbottom,pre_PWMFanb
ottom, tolerance);//flag



   neighborReadings= receiveFromNeighbor();//receive
data from neighbor. this is already a string

   sendToNeighbor(reading_list_string);//send data to
neighbor

  // compareNeighbor(reading_list);

   return(0);
}

char receiveFromNeighbor(){ //REMEMBER!!!! MC A will use
Serial1 to send, MC B used Serial2 to send. MC A listens
on Serial 2, MC B listens on Serial1 UPDATE THIS TO
REFLECT
  int i;

  if (Serial1.available()) {
    delay(100); //allows all serial sent to be received
together
    while(Serial1.available() && i<1) {
      i = Serial1.read();
    }
  neighborState(i);
    return(i);


}
}

int rangeCheck(int measured, int predefined, int
tolerance){
  int variable_flag;

  int range_max= predefined+tolerance*predefined;
  int range_min= predefined-tolerance*predefined;

  if(measured>range_max){
    variable_flag=1;
  }
  if(measured<range_min){
    variable_flag=1;
  }

  else{
    variable_flag=0;
  }

  return(variable_flag);


}

void sendToNeighbor(char readings){
  Serial1.write(State);
}

int identifyState(){
  int FailLimit=0;

  if(TempSensorHotFlag==1)
    {State=2;
    FailLimit=FailLimit+1;
      }
  else{State=1;}

  if(TempSensorColdFlag==1)
    {State=2;
      FailLimit=FailLimit+1;
      }
  else{State=1;}

  if(PeltierCubeVoltageFlag==1)
    {State=2;
      FailLimit=FailLimit+1;}
  else{State=1;}

  if(RPMFantopFlag==1)
    {State=2;}
  else{State=1;}
```

```
  if(RPMFanBottomFlag==1)
    {State=2;}
  else{State=1;}

  if(NeighborValuesFlag==1)
    {State=2;}
  else{State=1;}

  if(PWMFanTopFlag==1)
    {State=3;FailLimit=FailLimit+1;}
  else{State=1;}

  if(PWMFanbottomFlag==1)
    {State=3;FailLimit=FailLimit+1;}
  else{State=1;}

  if(FailLimit >= 3){
    State=3;
    return(0);
  }
}

int getSwitchState(){//0 is off, 1 is on
  int sw=analogRead(A8);

  return(sw);
}
void loop() {             //This function loops while
the arduino is powered
  watchDog();//if after 16 seconds of downtime the
system will auto restart

 int switch_position=getSwitchState();
  if(switch_position==0){
    return;
   }
  else
    {

  implementState(State);
  fanRPM_b;
  fanRPM_t;

  Standby();
  debugDisplay();
  delay(5000);
//  }

}
```