7565 code base

1. Main Loop
   1. Init control flow();
   2. Safe Call(); also does something
   3. …
   4. …………………………………………..while true……………………..……………………………..
   5. Handle line power() –

#define SAFE\_CALL(function, handler) ({control\_flow\_call(handler); function; control\_flow\_resume(handler);})

Firmware applications

# Application safety

STLCheckStack(void) – makes sure the stack doesn’t overflow

MaintainSafety(void)– resets watchdog timer, validates the timers, checks the control flow, bit compliments important variables, crc check, RAM test, CPU test, stack check

InitSafety(void) – initializes the safety protocols

# Holding Registers

# Input Registers

# Modbus handler

ReadCurrentStatus(uint8\_t \*buffer) -

Firmware drivers

# AC inputs

//important variable

acIrqData[NUM\_AC\_INPUTS] – large list with data for the AC IRQ

acResult[NUM\_AC\_INPUTS] – results structure shared between ISR and local thread

IsAcInputReady(void) – checks if any input has been sampled enough to process

HandleAC\_Inputs(void) – disables tim 14, saves the results to ‘localResults’ , sets the input struct back to false, then restarts tim 14

ACInputs\_HandleTimerIRQ(void) – runs ACInputs\_HandleIndividualAcInputIRQ(i) for each of the 5 AC inputs

ACInputs\_HandleIndividualAcInputIRQ(ACInputNames inputName) – reads the AC inputs and counts the samples for the main thread to process.

# Buttons

//important variables

\_\_STATIC bool buttonPressed

\_\_STATIC uint8\_t buttonDebounceCounts = 0;

HandleButton(void) – sets the InputBit from Io.h bit mask to tell if the button is pressed then checks to set the button pressed 1 second or 15 second bit. If button not pressed it resets the timers and bits.

ButtonHandleTimerTick(void) – handles button bounce, if button is pushed down it checks if it is at its MAX\_BUTTON\_DEBOUNCE\_CNTS, if not it ++ a counter and checks when its hit again, if its at the max it sets the button press to true. Same result for releasing the button.

# FlameLowLevel

//important variables

\_\_STATIC volatile uint16\_t flameSampleCount = 0 -

\_\_STATIC volatile uint16\_t flameSamplesISR[NUM\_FLAME\_SAMPLES]-buffer to hold samples

maxLight = 0; // maximum light reading over last 500ms (in lux)

minLight = 0; // minimum light reading over last 500ms (in lux)

avgLight = 0; // average light reading over last 500ms (in lux)

avgFlameOhms = 0; maxFlameOhms = 0; minFlameOhms = 0;

Several functions such as GetFlameMinLux(void) are used to get results to other places in the code.

ProcessADC\_Results(void) – paused tim14 to set a local variable for the flamesampleISR. Finds the average min and max values for light to get resistance (ohms). Uses analog to digital converter(ADC) to get min max average ohms.

RWB\_Activate\_ADC(void) – activates the analog to digital converter

Flame\_HandleDatapoint(uint16\_t reading) – sees when to take flame samples for saving.

# LEDs

//important variables

SINGLE\_LED = 1; DOUBLE\_LED = 2; bool toggleOn = false;

\_\_STATIC LedStruct strLEDs[NUM\_LEDS] = {…} – sets a struct for each state of the light along with the state of the control.

SetLedState(const LedName led, const LedState state)– sets the state/color of LED

LedState GetLedState(const LedName led) – returns the LED state

GetGreenLedCount(void) GetRedLedCount(void) – both check # of LEDs turned on

RWB\_ResetOutputPin(GPIO\_TypeDef \*port, uint32\_t pin)

RWB\_SetOutputPin(GPIO\_TypeDef \*port, uint32\_t pin)

RWB\_TogglePin(GPIO\_TypeDef \*port, uint32\_t pin) – changes the LED bit flag info

MaintainLeds(void) – shuts off LEDs at power loss and sets the state of the output pin for each Led to what color they should be, this is what actually turns on the LEDs, the other functions only set the value it should be.

# LinePower

GetLineCurrent(void) – returns current GetLineVoltage(void) – returns voltage

IsLineCurrentSettled(void) – checks if the current is stable by comparing a before and after current to see if its less than a set difference and returns bool value

ResetLineCurrentTimes(void) – resets timmers for taking current readings

LinePower\_HandleDatapoints(uint16\_t current\_adc, uint16\_t voltage\_adc) – takes the samples for line power

GetNextLowVoltageState(LowVoltageState currentState, const FactoryConfig \*config) – runs the low voltage state machine

CalculateLineVoltage(uint32\_t localSum, const FactoryConfig \*config) - yeah

CalculateLineCurrent(uint16\_t \*readings, const FactoryConfig \*config) - yeah

HandleLinePower(void) – copies the values from the ISR to the code, it also sets the powerloss timer when needed

ConvertADC\_ToVolts() – converts the line voltage from ADC to usable units for later need

GetLowVoltageState(void) - yeah

GetPowerLossDetected(void) - yeah

# MCU Temperatures

//important variables

isrMcuReadingSum= 0; isrMcuReadingCount = 0; used to read Micro controller temp

MCUTemperature\_HandleDatapoint(uint16\_t adc\_reading) – adds the temp reading to the data set till it get to 500 points

HandleMCUTemperature(void) – after 500 readings set the tempture and reset the readings

GetMCUTemperature(void) – returns the temp

# Outputs.c

Sets the output pins and clears them as needed

# RWB ADC

//important variables

bool adcReady = false;

ReadingStruct currentValues

uint16\_t adcReadingCount = 0;

bool adcOverflow = false; - error flags

RWBADC\_HandleTimerTick(void) – if acd is ready start the ADC conversion

RWBADC\_Setup(void) – clears the ADC1 flags and sets ADCReady to true

RWBADC\_IRQ(void) – check if sample or scanlist is ready and then preforms them

GetReferenceVoltage(void) – returns a set reference voltage

RefVoltage\_HandleDatapoints(uint16\_t adc\_reading) – checks if the voltage read is good and if not names it a bad sample and adds to the bad sample counter which shuts off the burner when threshold reached. The threshold is 4000 bad samples more than good samples.

# RWB firmware flash

//important variables

\_\_STATIC void \*volatile addrIsrFlashError = NULL – when there is a flash error, the NMI handler will set the errors address to this variable (NULL)

\_\_STATIC volatile uint32\_t addrIsrFlashErrorInv = UINT32\_MAX – set when address is not NULL

volatile uint8\_t flashCheckVerify = 0; - if set to magic variable then iit will issue a warning and the code will continue to execute till the watchdog timer times out

FirmwareFlashErrorHandler(uint32\_t idxErrorWord) – when an error occurs this sets the broken word to addrIsrFlashError then decides if the code should clear and continue or stop

FlashCheck(void) – scans each page in the flash memory and clears any page that has an error on it, and shuts down if needed.

# System bootloader jump

JumpToSystemBootloader(void) – jumps the code to the bootloader, by putting the call on the top of the stack. Then clears it. If it finishes the code then the controller will be shut down with a PC\_CORRUPTION error displayed.

# Timers

//important variables

\_\_STATIC TimerStruct strTimers[NUMTIMERS] – structure array that holds all the information for different timers. Set as time left, time stop, and cascade value which determines if it will auto roll into the next timer.

void RWB\_TIM#\_Init(void) – updates the timer and enable to counter

ResetTimer(TimerName timer) - yeah

GetTimer”value”(TimerName timer) – returns string of value

GetTimeRemaining(const TimerName timer) - yeah

GetElapsedTicks(const TimerName timer) – yeah

HandleIRQ\_TIM1(void) – interrupts every 10 ms and calls the button debounce function ButtonHandleTimerTick()

HandleIRQ\_TIM14(void) – interrupts every 0.5 ms to read the AC Inputs and trigger an ADC reading

HandleIRQ\_TIM16(void) – interrupt called for the uart timer

MaintainTimers(void) – checks that timers have updated since last time in loop, not quite sure what the rest does

ValidateTimerPresets(void) – check each of the timers with expected values and shut off the burner if they do not match up.

# Uart

\_\_STATIC UartMessageBuffer internalBuffer; - struct with state, size, and message

\_\_STATIC UartMessageBuffer sendBuffer; - struct with state, size, and message

\_\_STATIC uint16\_t sendIndex;

RWB\_Init\_Uart(void) – sets the buffers to have no data in the internal and external and initializes needed values

inline UartMessageBuffer \*GetMessageBuffer(void) – returns internal buffer struct

CanSendMessage(void) – checks that the buffers state is empty(NO\_MESSAGE)

SendMessage(uint8\_t \*data, uint16\_t size) – if you can send it and its larger than 0, and if the size is smaller than the max, then set the buffer to hold the message and then send it.

RWB\_Handle\_Uart\_IRQ(void) – checks if the tim16 interrupt is finished then resets it while running HandleRxIRQ();

RWB\_HandleUartTimerIRQ(void) – handles the interrupt by setting the internalbuffer.state to message\_ready

HandleRxIRQ(void) – saves the message to a receivedChar variable which is then transferred into the internal buffer body and setting the buffer to message in progress, this take one bit at a time and only adds 1 to the buffer during each iteration. Adds an error if the message gets bigger than the max buffer size.

HandleTxIRQ(void) – sets a few requirements for failure but if it passes it will send the message through the uart

ValidateUSARTSettings(void) – sets the baud rate and clock speed then checks that the expectedBRR matches the actualBRR, if not shut off the burner.

# Watchdog

ResetWatchdog(void) - yeah

Firmware Middleware

# AcInputAnalysis – TT, Lim, … terminals

//important variables

\_\_STATIC ACInputConfig acInputState[NUM\_AC\_INPUTS] = {} – NOT SURE

\_\_STATIC ACDiagData acDiags[NUM\_AC\_INPUTS][MAX\_AC\_DIAG\_RESULTS] = {};

\_\_STATIC uint32\_t acDiagsResultsIndex[NUM\_AC\_INPUTS] = {0, 0, 0, 0, 0};

IsAcInputOn(const ACInputResults result, float\_t percentHigh, float\_t percentLow) – asks if the result is valid, it’s within the range of required high and low samples, and the number of samples are okay. Checks to see if something is plugged into the AC input terminal.

ProcessInput(ACInputNames input, bool isActive) – verifies that something is plugged into the AC input terminal that can be used.

ShouldRecord(GenisysState state, ACInputNames input, bool isActive) - tells whether the system should start saving data from the AC input terminal if something is plugged in or report it to diagnostics.

HandleAcInputResults(const ACInputResults results[NUM\_AC\_INPUTS]) – for each input terminal if its ready, use them

GetACDiags(ACInputNames name, ACDiagData \*result) – makes space in memory for the results of the AC input

DisableLimitOnPowerLoss(void) – disables LIM if power is killed. By setting its bounce to 0 and clearing the input bit. This stops the flame earlier.

# AirProving

GetAirStatus(void) – determines the status of the Air, if its AIR\_NOT\_CONFIGURED, AIR\_BLOCKED\_FLUE, AIR\_NO\_COMBUSTION\_AIR, AIR\_OK

IsCapOk(void) – checks that the CAP is correct for both the 120 and 24 volt cap by seeing what state the motor is currently in.

IsBVOk(void) – checks the BV for both the 24 and 120 volts by seeing if its input bit has been set

IsAirOk(void) – checks that air is in the correct state by seeing if the air is either not configured or okay

HasAirProvingTimedOut(void) – check to see if air proving has timed out yet

ResetAirProvingCountdown(void) - Resets (extends) the time available for the system to prove air

HandleAirProvingTime(void) – keeps track of the timers and updates/resets them

# Cycle history

Yeah I’m good

# Flame

This one too

# Flame Diagnostics

//important variables

\_\_STATIC FlameCycleData currentFlame - struct with min, max, avg flame resistance

\_\_STATIC FlameHistoricData flameHistory – struct with standby avg and cycle data

\_\_STATIC uint8\_t flameDiagBufferIndex = 0;

\_\_STATIC FlameCycleData flameDiagBuffer[NUMBER\_OF\_DATAPOINTS\_IN\_HISTORY];

StoreFlameCycleData(FlameCycleData data) – adds the flame cycle data to a current variable and an array of variables using an index that gets updated until a max at 41 points.

FlameLoss(uint16\_t standbyFlame) -sotres the data from flame loss and what caused the event to occur

GetFlameCurrentData(void) – returns the current flame data

\*GetFlameHistory(void) – returns the address to the flame history

# flash Stack

I’m good

# Igniter

//important variables

\_\_STATIC IgniterState igniterState  = IGN\_OFF; - ignitor state begins off

\_\_STATIC uint16\_t igniterOffCurrent = 0xFFFF;

\_\_STATIC uint16\_t igniterCurrent    = 0xFFFF;

HandleIgniter(void) – checks if the output pin for the igniter is not set to turn it off and records the current state of the igniter.

RunIgniterStateMachine(IgniterState state) – several switch cases that run the igniter. Returns whatever state the igniter is in.

IGN\_MOTOR\_WAIT: waits for the motor to be ready and then resets timers

IGN\_RELAY\_WAIT: which waits for igniter relay to turn on then resets the line current readings

IGN\_CURRENT: waits for line current to settle and takes the samples

IGN\_ON: turns on igniter pin

IGN\_OFF: turns off igniter pin and safety shuts it down

GetIgniterState(void) – yeah

GetIgniterCurrent(void) - yeah

# Io

This function runs through and sets individual bits on a 8 bit bitmask by adding the values when set and clearing them when reset, this lets the value for the Enum equal 1 when set and 0 when false for all TT, Lim, flame, etc.

//important variables

uint16\_t inputBitmask            = 0; - used globally to keep track of inputs

uint16\_t inputBitmaskComplement  = 0xFFFF; - used to reset the input

uint16\_t outputBitmask           = 0; - used globally to keep track of outputs

uint16\_t outputBitmaskComplement = 0xFFFF; - used to reset the output

\_\_STATIC const uint8\_t IO\_BYTES  = 1

InputBit; - Enum with each bit such as flame, TT, Lim, button pressed, etc.

SetInputBit(InputBit input) – takes in input and or’s it with the inputbitmask(0), and and’s the compliment with the inputs compliment. This gicve you both the inputs regular and compliment bits

ClearInputBit(InputBit input) – sets the input bits oposit to the input bitmask and the compliment bitmask to the original

IsInputBitSet(InputBit input) – bool value checks if the input bitmask equals the input bit

void SetOutputBit(OutputBit output)

ClearOutputBit(OutputBit output)

IsOutputBitSet(OutputBit output)

IsCFH(void) – checks to see if there is a call for heat by comparing the current bitmask to the combination of both the TT and Lim bits on the bitmask.

IsButtonHeld(void) – compares the bitmask to the button held for 1 second and 15 seconds.

IsButtonEngaged(void) – same as button held but with button pressed

GetInputs(void) – return input bitmask

GetOutputs(void) – return output bitmask

IsButtonHeld\_15s(void) – checks if the 15s bit on the bitmask is set

# Life Time History

ResetCounters(void) – this resets every counter than can be reset, a wipe all

# Motor

//important variables

\_\_STATIC uint16\_t motorCurrent      = 0;

\_\_STATIC bool motorCurrentMeasured  = false;

\_\_STATIC uint16\_t ui16OffCurrent   = 0;

\_\_STATIC MotorState ui8MotorState  = MTR\_OFF; - start in motor off state

\_\_STATIC MotorSafetyRelayErrors motorError = SAFETY\_RELAY\_NO\_ERROR;

HandleMotor(void) – checks if motor ever needs to be turned off and sets the state to whatever state it should be in.

unMotorStateMachine(MotorState inputState) switch case for motor state machine taking in what state it should be in

case MTR\_STAGE1\_CHECK: checks for motor error and sets failure if occurred

case MTR\_SAFETY\_ON: resets current and check timers and proceeds to check #2

case MTR\_STAGE2\_CHECK: checks that line current didn’t increase after the relay turned on and will either kill the motor or set it to wait for the igniter.

case MTR\_WAIT\_FOR\_IGNITER: waits for both the igniter to be on and the line current to settle before moving onto the next state.

case MTR\_CHARGE\_CAP: runs MotorChargeCap(void) which shuts off the motor pin for 319 counts(0.4ms) before turning it back on.

case MTR\_TURNING\_ON: turns on motor pin and resets the state timer and motor check timer.

case MTR\_ON: just turns on motor pin.

case MTR\_TURN\_OFF: turns on safety relay and resets timers calls mtr safety off delay

case MTR\_SAFETY\_OFF\_DELAY: wats for the check timer to be ready and then shuts off the motor input bit. The checks that it shut off properly and calls MTR\_OFF

case MTR\_RELAY\_FAILURE\_TURN\_OFF: turns off motor pin, resets TMR\_MOTOR\_SAFETY\_OFF timer. Then calls a failure delay

case MTR\_RELAY\_FAILURE\_OFF\_DELAY:turn off motor,and wait for TMR\_MOTOR\_SAFETY\_OFF

timer to be finished before entering motor off state

case MTR\_STAGE\_1\_FAILURE: turns off motor pin and checks that the motor relay turns off properly.

case MTR\_OFF: turns off motor pin and constantly checks that the motor output bit is not set to turn on. Leaving only when that is called.

case MTR\_STAGE\_2\_OFF\_DELAY: another delay to keep motor off

case MTR\_STAGE\_3\_OFF\_DELAY: a third delay

default: shuts off motor with an error

GetMotorState(void) – returns state

MeasureMotorCurrent(void) – if line current is settled it will take a sample of the current and return true

GetMotorCurrent(void) – if its been measured returns the current, if not returns 0xFFFF

GetMotorCurrentMeasured(void) – returns motor current measured

\_\_STATIC \_\_attribute((optnone)) void MotorChargeCap(void) – stops the entire controller to wait 0.4 ms in order to charge the motor cap

# Presence

bool PresenceDetected(void) – return true if the TMR\_PRESENCE is not finished.

# Relay Check

//important variables

uint16\_t relayCheckStage x Current  = 0; one for each of the 3 stages + ending

RelayCheckResults relayCheckData;

uint16\_t MAX\_OFF\_CURRENT = 20;

RelayCheckStage1(void) – checks when all relays are open. gets line current and previous line current. Checks to see if it is greater than the max (returns SAFETY\_RELAY\_STAGE\_1\_RELAY\_WELD) checks the data and saves it all to a relay check data struct and saves stage, time, current, result, and safety relay set. Resets TMR\_RELAY\_CHECK\_DATA and returns last result.

RelayCheckStage2(void) – when safety relay is closed and all outputs are open. If line current settled, it will take a sample and compare it to the relay check 1, if that current difference is more than the max it trips an error, if not it will continue to save the data into the relay check data struct.

RelayCheckStage3(void) – checks if the safety input bit has been set and if the current is settled, the proceeds to check that the new delta line current between check 1 and 3 is not more than the max, then saves all of the data.

GetLastCurrentReading(void) – returns the last current reading

IsSafetyRelayError(MotorSafetyRelayErrors error) – returns if error is greater than 2 which is unknown.

GetLastRelayError(void) – returns last relay result

ResetRelayError(void) – sets the last result variable to NO\_ERROR

\*GetRelayCheckResults(void) – returns address to relay check data

# Safety

void ValidateBits(void \*s, void \*c, uint8\_t count) – takes in a non complimented and a complimented bit pointer, and the number of bits then determines if there is an error in the compliment. If they are different then the system will safety shutdown with the error COMPLEMENTED\_RAM\_CORRUPTION

# Status light

Has several functions for the purpose of running the lights depending on the status of the controller

# System config

Im good

# Valve

//important variables

\_\_STATIC uint16\_t valveCurrent    = 0xFFFF;

\_\_STATIC uint16\_t valveOffCurrent = 0xFFFF;

\_\_STATIC ValveState valveState    = VLV\_OFF; -starts with valve off

GetValveState(void) – returns valve state

HandleAutoValveEnable(uint16\_t currentDelta) – attempts to detect if a valve is connected to the controller and auto pair it with a pre purge and a post purge time. currentDelta is the difference of the current when the valve is off vs on to determine if one is connected.

This will run it through several cycle to try to connect but will not if the prepurge or postpurge is set to 0, if it has already tried to set one up, or if it has already been through too many cycles.

HandleValve(void) – if the output bit isn’t set, turn of valve, if it is run the state machine.

RunValveStateMachine(ValveState state) – ran by several switch statements for each state.

case VLV\_IGN\_WAIT: waits till the igniter is on then gets current and resets timers.

case VLV\_RELAY\_WAIT: wait for the valve relay to turn on then reset the line current.

case VLV\_CURRENT: if line current is settled then finds the valve current by subtracting the total system current from that. The sets then tried to auto configure a valve.

case VLV\_ON: turn the valve pin on

case VLV\_OFF: turn off valve pin, then constantly checks if the valve gets turned on to change the state to VLV\_IGN\_WAIT

default: safely shutdown the burner with the error UNEXPECTED\_DEFAULT\_CASE

GetValveCurrent(void) – returns the valve current

Game

Have Alfonzo chase you around and you have to collect his treats from hidden locations in my house, if he catches you he demands a treat very menacingly, kills you if you have none but lets you live in exchange for one.

Goal: find you phone wallet and keys and make it to the car

Cat logic: every second it checks his coordinates vs yours and determines if he should subtract/add to his x/y position in order to get closer to you, checking that the new position won’t be on top of an object. When you enter a new room you’ll have a set time in the room before he spawns (until logic gets better)