ME 507 Strobe Tuner

1

Generated by Doxygen 1.10.0

Chapter 1

ME 507 Strobe Tuner Documentation

Project Members: Jojo Penrose, Jared Sinasohn

Welcome to the ME 507 Strobe Tuner Documentation!

This manual provides an overview of the software side of our ME 507 Strobe Tuner project, including its features, setup, and usage instructions.

1.0.1 Overview and Theory

A strobe tuner is a highly accurate device used for tuning musical instruments by measuring the frequency of sound waves. Unlike conventional tuners, which often use needle or LED indicators to show pitch deviation, a strobe tuner operates by comparing the sound wave of the instrument to a reference frequency. It uses a rotating disk with accurately patterned slits cut into it, illuminated by a strobe light, to visually represent the difference between the played note and the desired pitch. As the disk spins, patterns of light and dark bands appear to move. When the frequency of the played note matches the reference frequency, the bands appear stationary, indicating that the instrument is in tune. If the instrument is out of tune, the bands will appear to move, depending on whether the pitch is flat or sharp.

However, in our research, we found that most strobe tuners can be thousands of dollars, which is an amount of money many musicians are unable to afford. So, our team set out to produce a strobe tuner running off a low-cost microcontroller with low cost parts to eventually make a low cost tuner that could one day be turned into a full product. We designed custom PCB's, created a full 3d-printed body, and coded in C to create an initial prototype. All in all, even though the initial prototype had many bugs, we were able to create a proof of concept showing our project is viable. If you would like to see more about the project, including information about the PCB Design, Analog filtering, mechanical design, and the trials and tribulations, visit `this link` which will take you to a Google Doc report about the project.

1.0.2 Features

Here are some of the features of our project:

- Asthetically pleasing exterior
- DC motor with full PIDF speed control
- I2C driven 16-segment display with sharp/flat LED's for display of the current tuning pitch
- Rotary knob encoder for custom pitch selection Here are some things we couldn't quite implement right now but will implement in the future:
- Input capture on the Microphone to auto-select tuning pitch
- Current sense on the motor through an ADC for over-current shutdown
- Making the I2C driver work

1.0.3 Getting Started in Software

All code for this project was completed in C using STM32CubeID (Cube). Our project runs on the STM32L476RGT6 chip, and before the code can be run it must be configured correctly using Cube's pin configuration GUI. The a table of the correct pin configurations for this project can be found below:

Category	Pin	Function	Description of Purpose	Notes
Serial Wire Debugging	PA13	SWD I/O	Serial wire Debugging via ST Link	N/A
	PA14	SWD Clock		
	PB3	SWD SWO		
GPIO	PA10	Output	Motor Enable Pin	N/A
	PB2	Output	Display Enable Pin	N/A
	PC10	Input	Auto/Manual Switch input	N/A
I2C	PB10	I2C2 SCL	I2C2 Bus for running the 16 segment display driver	20 KHz clock frequency
	PB11	I2C2 SDA		
UART	PA2	UART2 TX	UART for communicating to computer for debugging purposes	N/A
	PA3	UART2 RX		
ADC	PC5	ADC1 IN14	ADC for current sense of the motor	Single-ended
Timers	PA8	TIM1 CH2	Motor PWM Channels	CH1 CH2 set to PWM output, PSC = 0, ARR = 999
	PA9	TIM1 CH1		
	PA7	TIM3 CH1/CH2	Input Capture for the trigger of the RC Controller (Used for E-Stop)	CH1 Indirect Capture, rising edge detection, CH2 Direct Capture, falling edge detection, Enable NVIC Global Interrupts, PSC = 79, ARR = 65535
	PB0	TIM3 CH3/CH4	Input Capture for the wheel of the RC Controller	CH3 direct Capture, rising edge detection, CH4 indirect Capture, falling edge detection, Enable NVIC Global Interrupts, PSC = 79, ARR = 65536
	PB6	TIM4 CH1	Motor Encoder Quadrature Channels	Set timer to Encoder Mode, Capture on T1 and T2 PSC = 0, ARR = 65535
	PB7	TIM4 CH2		
	PA0	TIM5 CH1	Pitch Knob Rotary encoder Quadrature Channels	Capture on T1 and T2 PSC = 0, ARR = 1073741824
	PA1	TIM5 CH2		
	N/A	TIM6	Internal timer for speed calculations	N/A
	PC6	TIM8 CH1	Input Capture for microphone	Input Capture Direct Mode, PSC = 79, ARR = 65535, Rising Edge Detection, Enable Global NVIC interrupts
Crystal Oscillator	PH0	RCC OSC IN	Crystal Oscilator pins	80 MHZ internal clock
	PH1	RCC OSC OUT		

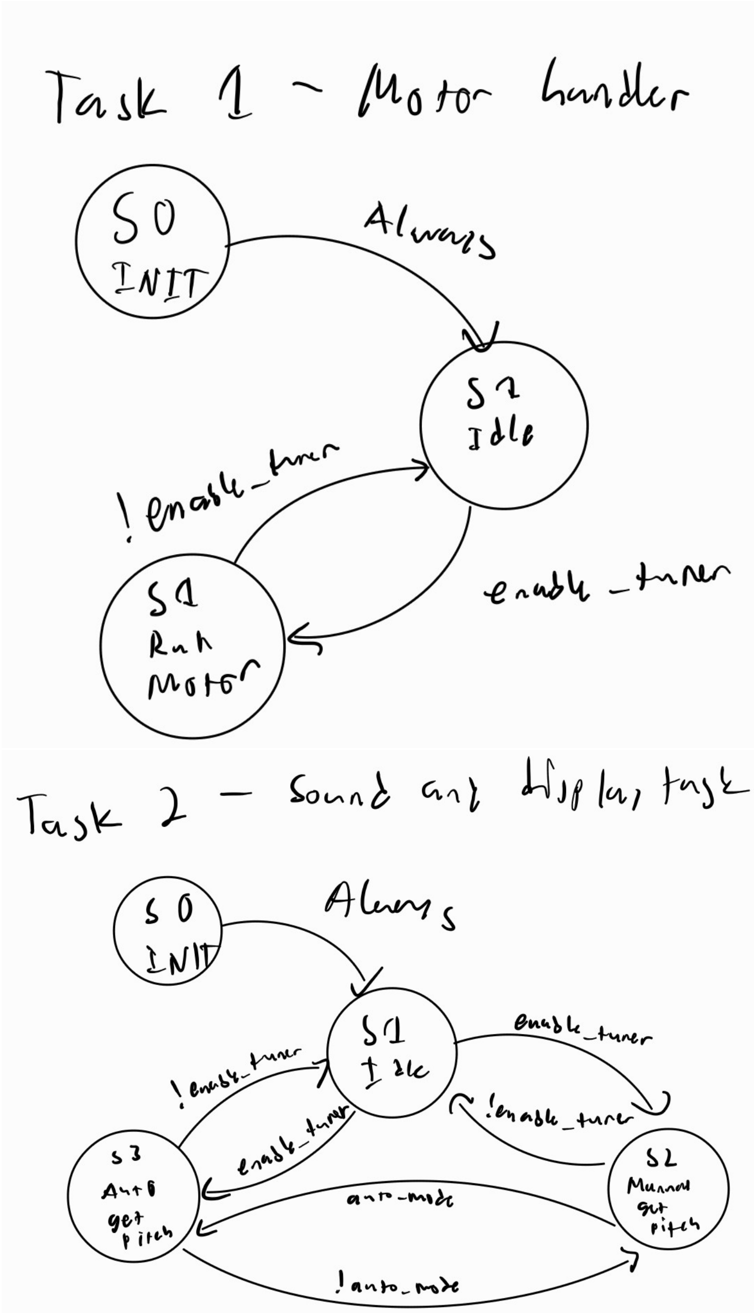
1.0.4 Software Implementation - Classes

Even though C does not support object oriented programming, an object oriented approact was taken when coding this project to streamline the coding process. Classes for a DC motor, RC_controller (not used but would have been used for E-stop), a Display, a Generic Encoder, a Pitch Selection encoder, an a Closed Loop Controller with full PIDF were written for the purposes of this project, and can be found below.

- CLController
- Display
- Encoder
- Motor
- PitchEncoder
- RC_Controller

1.0.5 Software Implementation - Cooperative Multitasking

In order to run all peripherals at once, a Cooperative Multitasking approach was taken. For this, different tasks for the microcontroller to execute were designed with each task handling a different function. The two tasks that were written are for handling pitch selection and the LED display, and the handling of the motor. Finite State machines were drawn for each task to explain how each task would operate, and images of these finite state machines can be found below.



Each task was run in a round-robin style, where each task assumes none of the other tasks will take a significant amount of time, allowing for each task to run simultaneously. This was deemed satisfactory for this project since there are only two tasks and each task is not all that complicated.

1.0.6 Conclusion

In this project, our team was able to create a proof of concept prototype for a mechanical strobe tuner. We are excited to continue this project in the future to make a low-cost strobe tuner that we can be proud of. Feel free to look through the documentation in this manual.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 CLController Class Reference

An implementation of a PIDF (F = Feedforward constant) closed loop controller. It is a generic controller that takes in a measurement, controller constants, and a setpoint and outputs an effort to be sent to the actuator.

```
#include <CLController.h>
```

Data Fields

- uint32_t kp
- uint32_t ki
- uint32_t kd
- uint32_t kf
- int32_t setpoint
- int32_t eff
- int32_t curr
- int32_t err
- int32_t err_acc
- uint8_t prev_err_index
- uint32_t initial_time
- uint32_t curr_time
- int32_t slope
- uint32_t prev_err_list_length
- int32_t prev_err_list [ ]

4.1.1 Detailed Description

An implementation of a PIDF (F = Feedforward constant) closed loop controller. It is a generic controller that takes in a measurement, controller constants, and a setpoint and outputs an effort to be sent to the actuator.

Parameters

kp	The propotional control constant multiplied by 10^6 for accuracy reasons
ki	The integral control constant multiplied by 10^6 for accuracy reasons
kd	The derivative control constant multiplied by 10^6 for accuracy reasons

Parameters

kf	The Feedforward constant multiplied by 10^6 for accuracy reasons. The feedforward constant allows for simpler speed control in this project.
setpoint	The reference input value of the controller
eff	The output effort calculated by the closed loop controller
curr	The currently measured value of the sensor in the loop
err	The error between the setpoint and measured values, used for proportional control
err_acc	How much error has been accumulating over time, used to estimate an integral control

Note

the following parameters all have to do with derivative control. Due to the nasty things noise can do to derivative control, a simple impulse filter has been implemented. It works by accumulating a list of error values and taking the slope vs time of that list once the list is full, and using that derivative list until the list fills again. This effectively filters out impulses due to the averaging, and thus creates a smoother, noise-free derivative control.

Parameters

prev_err_index	the index for the list of previous errors
initial_time	The initial time when the controller began calculating derivative error data
curr_time	The current time at which derivitive error has been calculated
slope	The overall slope of the list of derivative errors
prev_err_list_length	The number of error samples to be taken for the list, as chosen by the control programmer
prev_err_list[ ]	The list previous error values are stored in. @Attention this must be the same length as the previous_err_list_length

Definition at line 38 of file CLController.h.

4.1.2 Field Documentation

4.1.2.1 curr

```
int32_t curr
```

Definition at line 45 of file CLController.h.

4.1.2.2 curr_time

```
uint32_t curr_time
```

Definition at line 50 of file CLController.h.

4.1.2.3 eff

```
int32_t eff
```

Definition at line 44 of file CLController.h.

4.1.2.4 err

```
int32_t err
```

Definition at line 46 of file CLController.h.

4.1.2.5 err_acc

```
int32_t err_acc
```

Definition at line 47 of file CLController.h.

4.1.2.6 initial_time

```
uint32_t initial_time
```

Definition at line 49 of file CLController.h.

4.1.2.7 kd

```
uint32_t kd
```

Definition at line 41 of file CLController.h.

4.1.2.8 kf

```
uint32_t kf
```

Definition at line 42 of file CLController.h.

4.1.2.9 ki

```
uint32_t ki
```

Definition at line 40 of file CLController.h.

4.1.2.10 kp

```
uint32_t kp
```

Definition at line 39 of file CLController.h.

4.1.2.11 prev_err_index

```
uint8_t prev_err_index
```

Definition at line 48 of file CLController.h.

4.1.2.12 prev_err_list

```
int32_t prev_err_list[]
```

Definition at line 53 of file CLController.h.

4.1.2.13 prev_err_list_length

```
uint32_t prev_err_list_length
```

Definition at line 52 of file CLController.h.

4.1.2.14 setpoint

```
int32_t setpoint
```

Definition at line 43 of file CLController.h.

4.1.2.15 slope

```
int32_t slope
```

Definition at line 51 of file CLController.h.

The documentation for this class was generated from the following file:

- doxy_core/Inc/CLController.h

4.2 Display Class Reference

The class that implements the control of the TLC59116 16-Channel FM+ I2C-Bus Constant-Current LED Sink Driver to display the current note on the tuner.

```
#include <display_driver.h>
```

Data Fields

- I2C_HandleTypeDef * hi2c
- uint8_t curr_note
- USART_TypeDef * huart

4.2.1 Detailed Description

The class that implements the control of the TLC59116 16-Channel FM+ I2C-Bus Constant-Current LED Sink Driver to display the current note on the tuner.

Parameters

hi2c	The i2c bus the display driver is on
curr_note	The current note to display, a value from 0-11 mapping from A-Ab
huart	The uart bus to provide serial communication to a computer console for displaying the current note in debug note. In the final version, this parameter will be removed as it is unnecessary to run the actual display, but for now we need it to see what notes are being displayed to the console.

Definition at line 24 of file display_driver.h.

4.2.2 Field Documentation

4.2.2.1 curr_note

```
uint8_t curr_note
```

Definition at line 26 of file display_driver.h.

4.2.2.2 hi2c

```
I2C_HandleTypeDef* hi2c
```

Definition at line 25 of file display_driver.h.

4.2.2.3 huart

```
USART_TypeDef* huart
```

Definition at line 27 of file display_driver.h.

The documentation for this class was generated from the following file:

- doxy_core/Inc/display_driver.h

4.3 Encoder Class Reference

A generic implementation of a quadrature encoder that tracks the position of the encoder in counts and the speed the encoder is traveling at in counts per second.

```
#include <encoder_handler.h>
```

Data Fields

- TIM_HandleTypeDef * timer
- TIM_HandleTypeDef * timing_timer
- uint32_t prev_count
- uint32_t curr_count
- uint32_t prev_time
- uint32_t curr_time
- int32_t pos
- int32_t speed
- int32_t dx
- int32_t dt

4.3.1 Detailed Description

A generic implementation of a quadrature encoder that tracks the position of the encoder in counts and the speed the encoder is traveling at in counts per second.

Parameters

timer	The timer the encoder channels are on.
timing_timer	The timer that is used to track time in order to calculate speed accurately.

Attention

The timing_timer must be configures such that it counts values in microseconds

Parameters

prev_count	The previous count timer read.
curr_count	The current count timer reads.
prev_time	The previous count timing_timer read.
curr_time	The current count timing_timer reads.
pos	The position of the encoder in counts.
speed	The speed the motor is traveling at in counts per second
dx	The difference in position between the previous encoder reading and the current encoder reading in counts
dx	The difference in time between the previous encoder reading and the current encoder reading in microseconds

Definition at line 32 of file encoder_handler.h.

4.3.2 Field Documentation

4.3.2.1 curr_count

```
uint32_t curr_count
```

Definition at line 36 of file encoder_handler.h.

4.3.2.2 curr_time

```
uint32_t curr_time
```

Definition at line 38 of file encoder_handler.h.

4.3.2.3 dt

```
int32_t dt
```

Definition at line 42 of file encoder_handler.h.

4.3.2.4 dx

```
int32_t dx
```

Definition at line 41 of file encoder_handler.h.

4.3.2.5 pos

```
int32_t pos
```

Definition at line 39 of file encoder_handler.h.

4.3.2.6 prev_count

```
uint32_t prev_count
```

Definition at line 35 of file encoder_handler.h.

4.3.2.7 prev_time

```
uint32_t prev_time
```

Definition at line 37 of file encoder_handler.h.

4.3.2.8 speed

```
int32_t speed
```

Definition at line 40 of file encoder_handler.h.

4.3.2.9 timer

```
TIM_HandleTypeDef* timer
```

Definition at line 33 of file encoder_handler.h.

4.3.2.10 timing_timer

```
TIM_HandleTypeDef* timing_timer
```

Definition at line 34 of file encoder_handler.h.

The documentation for this class was generated from the following file:

- doxy_core/Inc/encoder_handler.h

4.4 Motor Class Reference

An implementation of a motor driver using a struct to emulate Object Oriented Programming. The motor has 4 parameters, timer which indicates the timer to be used to run the motor, channels, which indicates the channels to be used to run the motor, duty_cycle, the duty cycle to run the motor at, and enable_flag, which determines if the motor is allowed to run.

```
#include <motor_driver.h>
```

Data Fields

- TIM_HandleTypeDef * timer
- uint8_t channels
- int32_t duty_cycle
- uint8_t enable_flag

4.4.1 Detailed Description

An implementation of a motor driver using a struct to emulate Object Oriented Programming. The motor has 4 parameters, timer which indicates the timer to be used to run the motor, channels, which indicates the channels to be used to run the motor, duty_cycle, the duty cycle to run the motor at, and enable_flag, which determines if the motor is allowed to run.

Parameters

timer	The microcontroller timer to use to control the pwm in the motor. This is a timer pointer.
channels	The channels the PWM signal should run on. 1 corresponds to 1 and 2, and 2 corresponds to 3 and 4.
duty_cycle	The duty cycle of the motor from -ARR to ARR, which is 1000
enable_flag	The flag which indicates whether or not the motor should be allowed to run. A zero means the motor is disabled and a one means the motor is enabled.

Definition at line 29 of file motor_driver.h.

4.4.2 Field Documentation

4.4.2.1 channels

```
uint8_t channels
```

Definition at line 31 of file motor_driver.h.

4.4.2.2 duty_cycle

```
int32_t duty_cycle
```

Definition at line 32 of file motor_driver.h.

4.4.2.3 enable_flag

```
uint8_t enable_flag
```

Definition at line 33 of file motor_driver.h.

4.4.2.4 timer

```
TIM_HandleTypeDef* timer
```

Definition at line 30 of file motor_driver.h.

The documentation for this class was generated from the following file:

- doxy_core/Inc/motor_driver.h

4.5 PitchEncoder Class Reference

This class, in an object oriented sense, inherits the Encoder class but specifically reads values from the pitch encoder to map them to a specific pitch.

```
#include <pitch_encoder_handler.h>
```

Data Fields

- int16_t pitch
- Encoder * encoder
- int16_t delta

4.5.1 Detailed Description

This class, in an object oriented sense, inherits the Encoder class but specifically reads values from the pitch encoder to map them to a specific pitch.

Parameters

pitch	The current pitch the PitchEncoder read
encoder	The base encoder the PitchEncoder uses (This is equivalent to inheriting the encoder class in an object-oriented language)
delta	The difference between the previous reading of the encoder and the current reading of the encoder.

Definition at line 23 of file pitch_encoder_handler.h.

4.5.2 Field Documentation

4.5.2.1 delta

```
int16_t delta
```

Definition at line 26 of file pitch_encoder_handler.h.

4.5.2.2 encoder

```
Encoder* encoder
```

Definition at line 25 of file pitch_encoder_handler.h.

4.5.2.3 pitch

```
int16_t pitch
```

Definition at line 24 of file pitch_encoder_handler.h.

The documentation for this class was generated from the following file:

- doxy_core/Inc/pitch_encoder_handler.h

4.6 RC_Controller Class Reference

An implementation of an RC controller that sends pulse widths from its two control axes–a trigger and a wheel–through infrared. The motor has parameters for the timer used for the input capture of the pulse widths, the pulse widths sent by the controller for each of the axes, the values of time of the rising edges and falling axes for each axis, and flags that indicate a falling edge has been detected and therefore a period should be caulculated.

```
#include <controller_driver.h>
```

Data Fields

- TIM_HandleTypeDef * timer
- uint32_t period1
- uint32_t period2
- uint32_t re1
- uint32_t re2
- uint32_t fe1
- uint32_t fe2
- uint8_t fe_flag1
- uint8_t fe_flag2

4.6.1 Detailed Description

An implementation of an RC controller that sends pulse widths from its two control axes–a trigger and a wheel–through infrared. The motor has parameters for the timer used for the input capture of the pulse widths, the pulse widths sent by the controller for each of the axes, the values of time of the rising edges and falling axes for each axis, and flags that indicate a falling edge has been detected and therefore a period should be caulculated.

Parameters

timer	The microcontroller timer to use to run the input capture of the RC controller. This is a timer pointer. The timer should be configured such that each count of the timer is read in microseconds, which in this case requires a prescaler of 79 and an auto-reload of 65535.
period1	The pulse width of the first axis
period2	The pulse width of the second axis
re1	Timer value on the rising edge captured on the first axis
re2	Timer value on the rising edge captured on the second axis
fe1	Timer value on the falling edge captured on the first axis
fe2	Timer value on the falling edge captured on the second axis
fe_flag1	Flag that is set by the input capture callback function to indicate a falling edge has been detected for the first axis.
fe_flag2	Flag that is set by the input capture callback function to indicate a falling edge has been detected for the first axis.

Definition at line 33 of file controller_driver.h.

4.6.2 Field Documentation

4.6.2.1 fe1

```
uint32_t fe1
```

Definition at line 39 of file controller_driver.h.

4.6.2.2 fe2

```
uint32_t fe2
```

Definition at line 40 of file controller_driver.h.

4.6.2.3 fe_flag1

```
uint8_t fe_flag1
```

Definition at line 41 of file controller_driver.h.

4.6.2.4 fe_flag2

```
uint8_t fe_flag2
```

Definition at line 42 of file controller_driver.h.

4.6.2.5 period1

```
uint32_t period1
```

Definition at line 35 of file controller_driver.h.

4.6.2.6 period2

```
uint32_t period2
```

Definition at line 36 of file controller_driver.h.

4.6.2.7 re1

```
uint32_t re1
```

Definition at line 37 of file controller_driver.h.

4.6.2.8 re2

```
uint32_t re2
```

Definition at line 38 of file controller_driver.h.

4.6.2.9 timer

```
TIM_HandleTypeDef* timer
```

Definition at line 34 of file controller_driver.h.

The documentation for this class was generated from the following file:

- doxy_core/Inc/controller_driver.h

Chapter 5

File Documentation

5.1 doxy_core/Inc/CLController.h File Reference

This file defines the abstract class and methods for a closed-loop PIDF controller.

```
#include "stm32L4xx_hal.h"
```

Data Structures

- class CLController

  An implementation of a PIDF (F = Feedforward constant) closed loop controller. It is a generic controller that takes in a measurement, controller constants, and a setpoint and outputs an effort to be sent to the actuator.

Functions

- uint32_t run (CLController *con, int32_t measured)
- void reset_controller (CLController *con)

5.1.1 Detailed Description

This file defines the abstract class and methods for a closed-loop PIDF controller.

Date

May 29, 2024

Author

Jared Sinasohn

Definition in file CLController.h.

5.1.2 Function Documentation

5.1.2.1 reset_controller()

```
void reset_controller (
            CLController * con )
```

This function hard resets the controller (keeping controller gains the same)

Parameters

con	The CLController instance to run the plant with

Definition at line 50 of file CLController.c.

```
00050                                                         {
00051       con->err = 0;
00052       con->eff = 0;
00053       con->err_acc = 0;
00054       con->slope = 0;
00055       con->curr = 0;
00056       con->initial_time = HAL_GetTick();
00057       con->curr_time = HAL_GetTick();
00058 }
```

5.1.2.2 run()

```
uint32_t run (
            CLController * con,
            int32_t measured )
```

This function runs the Closed Loop Controller based on the parameters set up initially by the programmer

Parameters

con	The CLController instance to run the plant with
measured	The value measured by the sensor to be fed back into the controller

Returns

eff The effort/output calculated by the Controller to send to the actuator

This function runs the Closed Loop Controller based on the parameters set up initially by the programmer

Parameters

con	The CLController instance to run the plant with
measured	The value measured by the sensor to be fed back into the controller

Returns

eff The effort/output calculated by the Controller to send to the actuator

Attention

This function should be run at a rate slower than 1ms so as to not have zero time differential (if you are not using derivative control you can run at whatever speed you desire.)

store the sensor value as the current value

get the current time of simulation.

calculate the error between the setpoint and the measured value add this error to the accumulated error, but only if integral control has been implemented.

set the effort to be the proportional control plus integral control plus feed-forward control

if derivative control is enabled, add the current error to the list of errors and increment the index

if the list of errors is full, calculate the slope of the list and multiply it by the derivative constant and add it to the effort

Definition at line 18 of file CLController.c.

```
00018                                                     {
00020     con->curr = measured;
00022     con->curr_time = HAL_GetTick();
00024     con->err = con->setpoint - con->curr;
00026     if(con->ki > 0){
00027         con->err_acc = con->err_acc + con->err;
00028     }
00030     con->eff = (con->kf * con->setpoint)/1000000 + (con->kp * con->err)/1000000 + (con->ki *
      con->err_acc)/1000000;
00032     if(con->kd > 0){
00033         con->prev_err_list[con->prev_err_index] = con->err;
00034         con->prev_err_index += 1;
00035     }
00037     if(con->kd > 0 && con->prev_err_index >= con->prev_err_list_length){
00038         con->slope =
      ((con->prev_err_list[con->prev_err_index-1]-con->prev_err_list[0]))*1000/(con->curr_time -
      con->initial_time);
00039         con->eff += con->kd * con->slope;
00040         con->prev_err_index = 0;
00041         con->initial_time = con->curr_time;
00042     }
00043     return con->eff;
00044 }
```

5.2 CLController.h

Go to the documentation of this file.

```

#ifndef SRC_CLCONTROLLER_H_
#define SRC_CLCONTROLLER_H_

#include "stm32L4xx_hal.h"

typedef struct{
    uint32_t kp;
    uint32_t ki;
    uint32_t kd;
    uint32_t kf;
    int32_t setpoint;
    int32_t eff;
    int32_t curr;
    int32_t err;
    int32_t err_acc;
    uint8_t prev_err_index;
    uint32_t initial_time;
    uint32_t curr_time;
    int32_t slope;
    uint32_t prev_err_list_length;
    int32_t prev_err_list[];
}CLController;


uint32_t run(CLController* con,  int32_t measured);
void reset_controller(CLController* con);



#endif /* SRC_CLCONTROLLER_H_ */
```

5.3 doxy_core/Inc/controller_driver.h File Reference

A pseudo-object-oriented structure for implementing the calculation of a pulse-width read from a two-channel rc controller.

`#include "stm32l4xx_hal.h"`

Data Structures

- class RC_Controller

  An implementation of an RC controller that sends pulse widths from its two control axes–a trigger and a wheel–through infrared. The motor has parameters for the timer used for the input capture of the pulse widths, the pulse widths sent by the controller for each of the axes, the values of time of the rising edges and falling axes for each axis, and flags that indicate a falling edge has been detected and therefore a period should be caulculated.

Functions

- void controller_driver_calc_per1 (RC_Controller *controller)
- void controller_driver_calc_per2 (RC_Controller *controller)

5.3.1 Detailed Description

A pseudo-object-oriented structure for implementing the calculation of a pulse-width read from a two-channel rc controller.

Date

May 9, 2024

Author

Jared Sinasohn

Definition in file controller_driver.h.

5.3.2 Function Documentation

5.3.2.1 controller_driver_calc_per1()

```
void controller_driver_calc_per1 (
            RC_Controller * controller )
```

Calculates the pulse width sent by the first axis

Parameters

controller,the	RC Controller instance to caculate and operate on

Calculates the pulse width sent by the first axis. This number, if the timer is set up correctly, should be a number

Parameters

controller,the	RC Controller instance to caculate and operate on

Definition at line 15 of file controller_driver.c.

```
{
    uint32_t per1 = controller->fe1-controller->re1;
    if(per1 < 2050 && per1 > 950){
        controller->period1 = per1;
    }
}
```

5.3.2.2 controller_driver_calc_per2()

```
void controller_driver_calc_per2 (
            RC_Controller * controller )
```

Calculates the pulse width sent by the second axis

Parameters

controller,the	RC Controller instance to caculate and operate on

Calculates the pulse width sent by the first axis. This number, if the timer is set up correctly, should be a number

Parameters

controller,the	RC Controller instance to caculate and operate on

Definition at line 26 of file controller_driver.c.

```
00026                                                         {
00027     uint32_t per2 = controller->fe2-controller->re2;
00028         if(per2 < 2050 && per2 > 950){
00029             controller->period2 = per2;
00030         }
00031 }
```

5.4 controller_driver.h

Go to the documentation of this file.

```

#ifndef SRC_CONTROLLER_DRIVER_H_
#define SRC_CONTROLLER_DRIVER_H_


#include "stm32l4xx_hal.h"

typedef struct {
    TIM_HandleTypeDef* timer; // Handle to the HAL timer object
    uint32_t  period1; // pulse width of the first axis
    uint32_t  period2; // pulse width of the second axis
    uint32_t  re1; // Timer value on the rising edge captured on the first axis
    uint32_t  re2; // Timer value on the rising edge captured on the second axis
    uint32_t  fe1; // Timer value on the falling edge captured on the first axis
    uint32_t  fe2; // Timer value on the falling edge captured on the second axis
    uint8_t  fe_flag1; // flag that is set by the input capture callback function to indicate a
falling edge has been detected for the first axis.
    uint8_t  fe_flag2; // flag that is set by the input capture callback function to indicate a
falling edge has been detected for the first axis.
} RC_Controller;

void controller_driver_calc_per1(RC_Controller* controller);

void controller_driver_calc_per2(RC_Controller* controller);

#endif /* SRC_CONTROLLER_DRIVER_H_ */
```

5.5 doxy_core/Inc/display_driver.h File Reference

The header file for the i2c driven 16 segment display class.

```
#include "stm32l4xx_hal.h"
#include "stdio.h"
```

Data Structures

- class Display

  The class that implements the control of the TLC59116 16-Channel FM+ I2C-Bus Constant-Current LED Sink Driver to display the current note on the tuner.

Functions

- uint8_t display_note (Display ∗disp, uint8_t note)

5.5.1 Detailed Description

The header file for the i2c driven 16 segment display class.

Date

May 30, 2024

Author

Jared Sinasohn

Definition in file display_driver.h.

5.5.2 Function Documentation

5.5.2.1 display_note()

```
uint8_t display_note (
            Display * disp,
            uint8_t note )
```

This function displays the current note on the display.

Parameters

disp	The display instance
note,the	note to be displayed from 0-11.

Returns

curr_note The note that is displayed.

This function displays the current note on the display.

Parameters

disp	The display instance
note	The note to be displayed from 0-11.

Returns

curr_note The note that is displayed.

Attention

The Current function uses uart to display the current note, but at the end would use i2c to the driver.

Definition at line 28 of file display_driver.c.

```
00028                                                  {
00029     if(disp->curr_note == note){
00030         return disp->curr_note;
00031     }
00032     disp->curr_note = note;
00033     switch(disp->curr_note){
00034             case 0:
00035                 sprintf(Pitch_Buffer, "A  \r\n");
00036                 break;
00037             case 1:
00038                 sprintf(Pitch_Buffer, "Bb \r\n");
00039                 break;
00040             case 2:
00041                 sprintf(Pitch_Buffer, "B  \r\n");
00042                 break;
00043             case 3:
00044                 sprintf(Pitch_Buffer, "C  \r\n");
00045                 break;
00046             case 4:
00047                 sprintf(Pitch_Buffer, "Db \r\n");
00048                 break;
00049             case 5:
00050                 sprintf(Pitch_Buffer, "D  \r\n");
00051                 break;
00052             case 6:
00053                 sprintf(Pitch_Buffer, "Eb \r\n");
00054                 break;
00055             case 7:
00056                 sprintf(Pitch_Buffer, "E  \r\n");
00057                 break;
00058             case 8:
00059                 sprintf(Pitch_Buffer, "F  \r\n");
00060                 break;
00061             case 9:
00062                 sprintf(Pitch_Buffer, "Gb \r\n");
00063                 break;
00064             case 10:
00065                 sprintf(Pitch_Buffer, "G  \r\n");
00066                 break;
00067             case 11:
00068                 sprintf(Pitch_Buffer, "Ab \r\n");
00069                 break;
00070         }
00071         HAL_UART_Transmit(disp->huart, Pitch_Message, sizeof(Pitch_Message), 10000);
00072         HAL_UART_Transmit(disp->huart, Pitch_Buffer, sizeof(Pitch_Buffer), 10000);
00073     return disp->curr_note;
00074 }
```

5.6 display_driver.h

Go to the documentation of this file.

```

#ifndef SRC_DISPLAY_DRIVER_H_
#define SRC_DISPLAY_DRIVER_H_

#include "stm32l4xx_hal.h"
#include "stdio.h"

typedef struct{
    I2C_HandleTypeDef* hi2c;
    uint8_t curr_note;
    USART_TypeDef * huart;
}Display;

uint8_t display_note(Display* disp, uint8_t note);
#endif /* SRC_DISPLAY_DRIVER_H_ */
```

5.7 doxy_core/Inc/encoder_handler.h File Reference

The header file for a class that implements a generic quadrature encoder.

`#include "stm32L4xx_hal.h"`

Data Structures

- class Encoder

  A generic implementation of a quadrature encoder that tracks the position of the encoder in counts and the speed the encoder is traveling at in counts per second.

Functions

- void encoder_read_curr_state (Encoder ∗encoder)
- int32_t encoder_calc_speed (Encoder ∗encoder, int32_t dx, int32_t dt)
- void zero (Encoder ∗encoder)
- int32_t delta (TIM_HandleTypeDef ∗timer, uint32_t initial, uint32_t final)

5.7.1 Detailed Description

The header file for a class that implements a generic quadrature encoder.

Date

May 18, 2024

Author

Jared Sinasohn

Definition in file encoder_handler.h.

5.7.2 Function Documentation

5.7.2.1 delta()

```
int32_t delta (
            TIM_HandleTypeDef * timer,
            uint32_t initial,
            uint32_t final )
```

This function calculates a delta between two values from a timer, while accounting for overflow.

Parameters

timer	The timer the initial and final values come from
initial	The initial value of the timer
final	The final value of the timer

Returns

delta The corrected delta value to account for overflow and underflow.

get the auto reload value of the timer since it is the maximum value a timer can be

we can determine if something has overflowed or underflowed be assuming a delta will never be greater than half the auto reload value, which if the encoder is read enough is a good assumption

if the value underflows, the delta will be a positive value greater than overflow, so just subtract off ARR+1 from the underflowed delta

similarly, if the value overflows, the delta will be a negative value less than negative of overerflow, so add ARR+1 to the overflowed delta

Definition at line 71 of file encoder_handler.c.

```
00071                                                                    {
00073     uint32_t ARR = (int32_t)(timer->Init.Period );
00075     int32_t overflow = ((ARR-1)/2)+1;
00076     //1 Calculate the delta
00077     int32_t delta = final-initial;
00079     if(delta >= overflow){
00080         delta = delta - overflow*2;
00082     }else if(delta <= -1*overflow){
00083         delta = delta + overflow;
00084     }
00085     return delta;
00086 }
```

5.7.2.2 encoder_calc_speed()

```
int32_t encoder_calc_speed (
            Encoder * encoder,
            int32_t dx,
            int32_t dt )
```

This function calculates a speed based on a differential position and a differential time.

Parameters

encoder	The encoder instance to operate on
dx	The differential position term in encoder counts.
dt	The differential time term in microseconds

Returns

The calculated speed in counts per second

if the delta time is zero, return the previous speed to avoid divide by zero errors.

return the speed in counts/second knowing that dt is in microseconds

Definition at line 39 of file encoder_handler.c.

```
00039                                                                  {
00041     if(dt == 0){
00042         return encoder->speed;
00043     }
00045     return ((dx)*1000000)/dt;
00046 }
```

5.7.2.3 encoder_read_curr_state()

```
void encoder_read_curr_state (
            Encoder * encoder )
```

This function reads the current encoder state and uses it to calculate the speed and position of the encoder.

Parameters

encoder	The encoder instance to operate on.

Attention

If the timing_timer is set up correctly for the inputted encoder, this function should be called no faster than every microsecond.

set the previous times and previous counts to the previously current values

get the count and time values from the two timers

calculate the difference between the current counts/times and previous counts/times using the delta function

set the encoder position to be the previous encoder position plus the delta position

set the speed of the encoder to the calculated value via encoder_calc_speed()

Definition at line 15 of file encoder_handler.c.

```
00015                                                {
00017     encoder->prev_count = encoder->curr_count;
00018     encoder->prev_time = encoder->curr_time;
00020     encoder->curr_count = encoder->timer->Instance->CNT;
00021     encoder->curr_time = encoder->timing_timer->Instance->CNT;
00023     encoder->dx = delta(encoder->timer, encoder->prev_count,encoder->curr_count);
00024     encoder->dt = delta(encoder->timing_timer, encoder->prev_time,encoder->curr_time);
00026     encoder->pos = encoder->pos + encoder->dx;
00028     encoder->speed = encoder_calc_speed(encoder,encoder->dx,encoder->dt);
00029 
00030 }
```

5.7.2.4 zero()

```
void zero (
            Encoder * encoder )
```

This function zeros the encoder, including the timing_timer and timer timers.

Parameters

encoder	The encoder instance to operate on

Definition at line 52 of file encoder_handler.c.

```
00052                            {
00053     encoder->timer->Instance->CNT = 0;
00054     encoder->timing_timer->Instance->CNT = 0;
00055     encoder->prev_count = 0;
00056     encoder->curr_count = 0;
00057     encoder->prev_time = 0;
00058     encoder->curr_time = 1;
00059     encoder->pos = 0;
00060     encoder->speed = 0;
00061 }
```

5.8 encoder_handler.h

Go to the documentation of this file.

```

#ifndef SRC_ENCODER_HANDLER_H_
#define SRC_ENCODER_HANDLER_H_

#include "stm32L4xx_hal.h"


typedef struct{
    TIM_HandleTypeDef* timer;
    TIM_HandleTypeDef* timing_timer;
    uint32_t prev_count;
    uint32_t curr_count;
    uint32_t prev_time;
    uint32_t curr_time;
    int32_t pos;
    int32_t speed;
    int32_t dx;
    int32_t dt;
}Encoder;

void encoder_read_curr_state(Encoder* encoder);

int32_t encoder_calc_speed(Encoder* encoder,int32_t dx, int32_t dt);

void zero(Encoder* encoder);

int32_t delta(TIM_HandleTypeDef* timer, uint32_t initial, uint32_t final);

#endif /* SRC_ENCODER_HANDLER_H_ */
```

5.9 doxy_core/Inc/main.h File Reference

: Header for main.c file. This file contains the common defines of the application.

`#include "stm32l4xx_hal.h"`

Macros

- #define USART_TX_Pin GPIO_PIN_2
- #define USART_TX_GPIO_Port GPIOA
- #define USART_RX_Pin GPIO_PIN_3
- #define USART_RX_GPIO_Port GPIOA
- #define TMS_Pin GPIO_PIN_13
- #define TMS_GPIO_Port GPIOA
- #define TCK_Pin GPIO_PIN_14
- #define TCK_GPIO_Port GPIOA

Functions

- void HAL_TIM_MspPostInit (TIM_HandleTypeDef *htim)
- void Error_Handler (void)

    This function is executed in case of error occurrence.

5.9.1 Detailed Description

: Header for main.c file. This file contains the common defines of the application.

!

Attention

Definition in file main.h.

5.9.2 Macro Definition Documentation

5.9.2.1 TCK_GPIO_Port

```
#define TCK_GPIO_Port GPIOA
```

Definition at line 69 of file main.h.

5.9.2.2 TCK_Pin

```
#define TCK_Pin GPIO_PIN_14
```

Definition at line 68 of file main.h.

5.9.2.3 TMS_GPIO_Port

```
#define TMS_GPIO_Port GPIOA
```

Definition at line 67 of file main.h.

5.9.2.4 TMS_Pin

```
#define TMS_Pin GPIO_PIN_13
```

Definition at line 66 of file main.h.

5.9.2.5 USART_RX_GPIO_Port

```
#define USART_RX_GPIO_Port GPIOA
```

Definition at line 65 of file main.h.

5.9.2.6 USART_RX_Pin

```
#define USART_RX_Pin GPIO_PIN_3
```

Definition at line 64 of file main.h.

5.9.2.7 USART_TX_GPIO_Port

```
#define USART_TX_GPIO_Port GPIOA
```

Definition at line 63 of file main.h.

5.9.2.8 USART_TX_Pin

```
#define USART_TX_Pin GPIO_PIN_2
```

Definition at line 62 of file main.h.

5.9.3 Function Documentation

5.9.3.1 Error_Handler()

```
void Error_Handler (
            void )
```

This function is executed in case of error occurrence.

Return values

None	

Definition at line 1054 of file main.c.

```
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}
```

5.9.3.2 HAL_TIM_MspPostInit()

```
void HAL_TIM_MspPostInit (
```

```
TIM_HandleTypeDef * htim )
```

5.10 main.h

Go to the documentation of this file.

```
/* USER CODE BEGIN Header */
/* USER CODE END Header */

/* Define to prevent recursive inclusion -------------------------------------*/
#ifndef __MAIN_H
#define __MAIN_H

#ifdef __cplusplus
extern "C" {
#endif

/* Includes ------------------------------------------------------------------*/
#include "stm32l4xx_hal.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Exported types ------------------------------------------------------------*/
/* USER CODE BEGIN ET */

/* USER CODE END ET */

/* Exported constants --------------------------------------------------------*/
/* USER CODE BEGIN EC */

/* USER CODE END EC */

/* Exported macro ------------------------------------------------------------*/
/* USER CODE BEGIN EM */

/* USER CODE END EM */

void HAL_TIM_MspPostInit(TIM_HandleTypeDef *htim);

/* Exported functions prototypes ---------------------------------------------*/
void Error_Handler(void);

/* USER CODE BEGIN EFP */

/* USER CODE END EFP */

/* Private defines -----------------------------------------------------------*/
#define USART_TX_Pin GPIO_PIN_2
#define USART_TX_GPIO_Port GPIOA
#define USART_RX_Pin GPIO_PIN_3
#define USART_RX_GPIO_Port GPIOA
#define TMS_Pin GPIO_PIN_13
#define TMS_GPIO_Port GPIOA
#define TCK_Pin GPIO_PIN_14
#define TCK_GPIO_Port GPIOA

/* USER CODE BEGIN Private defines */

/* USER CODE END Private defines */

#ifdef __cplusplus
}
#endif

#endif /* __MAIN_H */
```

5.11 doxy_core/Inc/motor_driver.h File Reference

The header file for a generic DC motor driven from a standard h-bridge motor driver.

```
#include "stm32l4xx_hal.h"
```

Data Structures

- class Motor

  An implementation of a motor driver using a struct to emulate Object Oriented Programming. The motor has 4 parameters, timer which indicates the timer to be used to run the motor, channels, which indicates the channels to be used to run the motor, duty_cycle, the duty cycle to run the motor at, and enable_flag, which determines if the motor is allowed to run.

Functions

- void motor_set_duty_cycle (Motor ∗motor, int32_t doot)
- void motor_enable_disable (Motor ∗motor, uint8_t enable)

5.11.1 Detailed Description

The header file for a generic DC motor driven from a standard h-bridge motor driver.

Date

Apr 25, 2024

Author

Jared Sinasohn

Definition in file motor_driver.h.

5.11.2 Function Documentation

5.11.2.1 motor_enable_disable()

```
void motor_enable_disable (
            Motor * motor,
            uint8_t enable )
```

Enables or disables motor based on user input

Parameters

motor,the	Motor struct to act upon
enable,the	boolean of whether to enable or disable the motor with 1 being to enable and 0 being to disable.

Definition at line 87 of file motor_driver.c.

```
00087                                                          {
00088     // if user wants to enable motor
00089     if(enable == 1){
00090         motor->enable_flag = 1;
00091         // First retrieve ARR to set motor to brake mode
00092         uint32_t ARR = (uint32_t)(motor->timer->Init.Period + 1);
00093 
00094         // Now set the correct motor pair to brake mode.
```

```
        if(motor->channels == 1){
            motor->timer->Instance->CCR1 = ARR;
            motor->timer->Instance->CCR2 = ARR;
        } else if(motor->channels == 2){
            motor->timer->Instance->CCR3 = ARR;
            motor->timer->Instance->CCR4 = ARR;
        }else{
            return;
        }

        // set the motor's enable flag to 1
        motor->enable_flag = 1;

    // if user wants to disable motor
    } else if(enable == 0){
        motor->enable_flag = 0;
        if(motor->channels == 1){
            motor->timer->Instance->CCR1 = 0;
            motor->timer->Instance->CCR2 = 0;
        } else if(motor->channels == 2){
            motor->timer->Instance->CCR3 = 0;
            motor->timer->Instance->CCR4 = 0;
        }else{
            return;
        }

        // set the motor's enable flag to 0
        motor->enable_flag = 0;
    }
}
```

5.11.2.2 motor_set_duty_cycle()

```
void motor_set_duty_cycle (
              Motor * motor,
              int32_t doot )
```

This function implements the duty cycle setting of the motor. It takes in the motor struct and a duty cycle from -100 to 100 (though the function saturates values above and below these values).

Parameters

motor,the	Motor struct to be operated on.
doot,the	duty cycle to be set to.

Definition at line 21 of file motor_driver.c.

```
00021                                                     {
00022     motor->duty_cycle = doot;
00023     // First, check if the motor is disabled
00024     if(motor->enable_flag != 1){
00025         // if the enable flag isn't set exit the function and do nothing.
00026         // we are also using != 1 so if there is a stray value in memory,
00027         // the motor doesn't accidentally enable.
00028         return;
00029     }
00030
00031     // Next, saturate the duty cycle just in case.
00032     if(doot < -100){
00033         doot = -100;
00034     }
00035     if(doot > 100){
00036         doot = 100;
00037     }
00038
00039     // We need to get the auto reload value for the timer we are using
00040     // signed value so we don't run into sign issues later
00041     int32_t ARR = (int32_t)(motor->timer->Init.Period + 1);
00042
00043     // Now calculate the duty cycle in terms of the CCR value
00044     doot = doot*ARR/100;  // multiply first so we don't lose data
00045
00046     // now we need to set the motors to the correct duty cycles
00047     // Forwards will be channels 1 and 3 for motors 1 and 2 respectively
00048     // Backwards will be channels 2 and 4 for motors 1 and 2 respectively
```

```


    // the below CCR's are based on the logic table of the toshiba, setting motor.
    // to brake mode
    // if duty cycle is <0
    if (doot < 0){
        // check if it is the first or second motor.
        if(motor->channels == 1){
            motor->timer->Instance->CCR1 = ARR;
            motor->timer->Instance->CCR2 = ARR + doot;
        } else if(motor->channels == 2){
            motor->timer->Instance->CCR3 = ARR;
            motor->timer->Instance->CCR4 = ARR + doot;
        }else{
            // if neither return
            return;
        }
    // if duty cycle >=0
    } else{
        if(motor->channels == 1){
            motor->timer->Instance->CCR1 = ARR - doot;
            motor->timer->Instance->CCR2 = ARR;
        } else if(motor->channels == 2){
            motor->timer->Instance->CCR3 = ARR - doot;
            motor->timer->Instance->CCR4 = ARR;
        }else{
            return;
        }
    }
}
```

5.12 motor_driver.h

Go to the documentation of this file.

```

#ifndef SRC_MOTOR_DRIVER_H_
#define SRC_MOTOR_DRIVER_H_

//include hal library
#include "stm32l4xx_hal.h"

typedef struct {
    TIM_HandleTypeDef* timer; // Handle to the HAL timer object
    uint8_t  channels; // which channels to use.  using channels 1-4 so 1 corresponds to 1 and 2 and 2 corresponds to 3 and 4
    int32_t  duty_cycle; // duty cycle of the motor from -ARR to ARR, which is 1000
    uint8_t  enable_flag; // flag that enables the motor
} Motor;

void motor_set_duty_cycle(Motor* motor, int32_t doot);

void motor_enable_disable(Motor* motor, uint8_t enable);


#endif /* SRC_MOTOR_DRIVER_H_ */
```

5.13 doxy_core/Inc/pitch_encoder_handler.h File Reference

```
#include "stm32L4xx_hal.h"
#include "encoder_handler.h"
```

Data Structures

- class PitchEncoder

  This class, in an object oriented sense, inherits the Encoder class but specifically reads values from the pitch encoder to map them to a specific pitch.

Functions

- uint32_t get_pitch (PitchEncoder *p_enc)

5.13.1 Detailed Description

Date

May 30, 2024

Author

Jared Sinasohn

Definition in file pitch_encoder_handler.h.

5.13.2 Function Documentation

5.13.2.1 get_pitch()

```
uint32_t get_pitch (
            PitchEncoder * p_enc )
```

This function gets the current pitch based on the pitch selection knob

Parameters

p_enc	The pitch encoder object to read from

Returns

the current pitch, a number 0-11 mapped through the chromatic notes from A-Ab

read the current state of the encoder

store the delta of the encoder

add the delta to the pitch

if the pitch hasn't changed, just return the pitch

we can treat the pitch as a number between 0 and 11, which can underflow and overflow. We can run a similar algorithm to the delta() function in the encoder class to correct for this.

Definition at line 15 of file pitch_encoder_handler.c.

```
00015                                          {
00017     encoder_read_curr_state((p_enc->encoder));
00019     p_enc->delta = (int16_t)(p_enc->encoder->dx);
00021     p_enc->pitch += p_enc->delta;
00023     if(p_enc->delta == 0){
00024         return p_enc->pitch;
00025     }
00027     if(p_enc->pitch < 0){
00028         p_enc->pitch += 12;
00029     }
00030     if(p_enc->pitch >= 12){
00031         p_enc->pitch -= 12;
00032     }
00033     return p_enc->pitch;
00034 }
```

5.14 pitch_encoder_handler.h

Go to the documentation of this file.

```
#ifndef SRC_PITCH_ENCODER_HANDLER_H_
#define SRC_PITCH_ENCODER_HANDLER_H_

#include "stm32L4xx_hal.h"
#include "encoder_handler.h"

typedef struct{
    int16_t pitch;
    Encoder* encoder;
    int16_t delta;
}PitchEncoder;

uint32_t get_pitch(PitchEncoder* p_enc);

#endif /* SRC_PITCH_ENCODER_HANDLER_H_ */
```

5.15 doxy_core/README.md File Reference

5.16 doxy_core/Src/CLController.c File Reference

this file implements the methods for the CLController class.

```
#include "CLController.h"
```

Functions

- uint32_t run (CLController *con, int32_t measured)
- void reset_controller (CLController *con)

5.16.1 Detailed Description

this file implements the methods for the CLController class.

Date

May 29, 2024

Author

Jared Sinasohn

Definition in file CLController.c.

5.16.2 Function Documentation

5.16.2.1 reset_controller()

```
void reset_controller (
        CLController * con )
```

This function hard resets the controller (keeping controller gains the same)

Parameters

con	The CLController instance to run the plant with

Definition at line 50 of file CLController.c.

```
00050                                           {
00051     con->err = 0;
00052     con->eff = 0;
00053     con->err_acc = 0;
00054     con->slope = 0;
00055     con->curr = 0;
00056     con->initial_time = HAL_GetTick();
00057     con->curr_time = HAL_GetTick();
00058 }
```

5.16.2.2 run()

```
uint32_t run (
        CLController * con,
        int32_t measured )
```

This function runs the Closed Loop Controller based on the parameters set up initially by the programmer

Parameters

con	The CLController instance to run the plant with
measured	The value measured by the sensor to be fed back into the controller

Returns

eff The effort/output calculated by the Controller to send to the actuator

Attention

This function should be run at a rate slower than 1ms so as to not have zero time differential (if you are not using derivative control you can run at whatever speed you desire.)

store the sensor value as the current value

get the current time of simulation.

calculate the error between the setpoint and the measured value

add this error to the accumulated error, but only if integral control has been implemented.

set the effort to be the proportional control plus integral control plus feed-forward control

if derivative control is enabled, add the current error to the list of errors and increment the index

if the list of errors is full, calculate the slope of the list and multiply it by the derivative constant and add it to the effort

Definition at line 18 of file CLController.c.

```
00018                                                    {
00020     con->curr = measured;
00022     con->curr_time = HAL_GetTick();
00024     con->err = con->setpoint - con->curr;
00026     if(con->ki > 0){
```

```
        con->err_acc = con->err_acc + con->err;
    }
    con->eff = (con->kf * con->setpoint)/1000000 + (con->kp * con->err)/1000000 + (con->ki * con->err_acc)/1000000;
    if(con->kd > 0){
        con->prev_err_list[con->prev_err_index] = con->err;
        con->prev_err_index += 1;
    }
    if(con->kd > 0 && con->prev_err_index >= con->prev_err_list_length){
        con->slope = ((con->prev_err_list[con->prev_err_index-1]-con->prev_err_list[0]))*1000/(con->curr_time - con->initial_time);
        con->eff += con->kd * con->slope;
        con->prev_err_index = 0;
        con->initial_time = con->curr_time;
    }
    return con->eff;
}
```

5.17 CLController.c

Go to the documentation of this file.

```

#include "CLController.h"

uint32_t run(CLController* con,  int32_t measured){
    con->curr = measured;
    con->curr_time = HAL_GetTick();
    con->err = con->setpoint - con->curr;
    if(con->ki > 0){
        con->err_acc = con->err_acc + con->err;
    }
    con->eff = (con->kf * con->setpoint)/1000000 + (con->kp * con->err)/1000000 + (con->ki * con->err_acc)/1000000;
    if(con->kd > 0){
        con->prev_err_list[con->prev_err_index] = con->err;
        con->prev_err_index += 1;
    }
    if(con->kd > 0 && con->prev_err_index >= con->prev_err_list_length){
        con->slope = ((con->prev_err_list[con->prev_err_index-1]-con->prev_err_list[0]))*1000/(con->curr_time - con->initial_time);
        con->eff += con->kd * con->slope;
        con->prev_err_index = 0;
        con->initial_time = con->curr_time;
    }
    return con->eff;
}
void reset_controller(CLController* con){
    con->err = 0;
    con->eff = 0;
    con->err_acc = 0;
    con->slope = 0;
    con->curr = 0;
    con->initial_time = HAL_GetTick();
    con->curr_time = HAL_GetTick();
}
```

5.18 doxy_core/Src/controller_driver.c File Reference

This file implements the mehtods to calculate the period for the RC_Controller class.

```
#include "controller_driver.h"
```

Functions

- void controller_driver_calc_per1 (RC_Controller ∗controller)
- void controller_driver_calc_per2 (RC_Controller ∗controller)

5.18.1 Detailed Description

This file implements the mehtods to calculate the period for the RC_Controller class.

Date

May 9, 2024

Author

Jared Sinasohn

Definition in file controller_driver.c.

5.18.2 Function Documentation

5.18.2.1 controller_driver_calc_per1()

```
void controller_driver_calc_per1 (
            RC_Controller * controller )
```

Calculates the pulse width sent by the first axis. This number, if the timer is set up correctly, should be a number

Parameters

controller,the	RC Controller instance to caculate and operate on

Definition at line 15 of file controller_driver.c.

```
00015                                                            {
00016     uint32_t per1 = controller->fe1-controller->re1;
00017     if(per1 < 2050 && per1 > 950){
00018         controller->period1 = per1;
00019     }
00020 }
```

5.18.2.2 controller_driver_calc_per2()

```
void controller_driver_calc_per2 (
            RC_Controller * controller )
```

Calculates the pulse width sent by the first axis. This number, if the timer is set up correctly, should be a number

Parameters

controller,the	RC Controller instance to caculate and operate on

Definition at line 26 of file controller_driver.c.

```
00026                                                            {
00027     uint32_t per2 = controller->fe2-controller->re2;
00028         if(per2 < 2050 && per2 > 950){
00029             controller->period2 = per2;
00030         }
00031 }
```

5.19 controller_driver.c

Go to the documentation of this file.

```
#include "controller_driver.h"

void controller_driver_calc_per1(RC_Controller* controller){
    uint32_t per1 = controller->fe1-controller->re1;
    if(per1 < 2050 && per1 > 950){
        controller->period1 = per1;
    }
}
void controller_driver_calc_per2(RC_Controller* controller){
    uint32_t per2 = controller->fe2-controller->re2;
        if(per2 < 2050 && per2 > 950){
            controller->period2 = per2;
        }
}
```

5.20 doxy_core/Src/display_driver.c File Reference

Implements the methods from the Display class.

```
#include "display_driver.h"
```

Functions

- uint8_t display_note (Display *disp, uint8_t note)

Variables

- uint16_t note_addresses [12] = {0b1101100001111000,0b1101101101110101,0b1101101101110100,0b1101101100000000,0
- uint16_t disp_addr = 0b11000001
- uint8_t Pitch_Message [ ] = "Current Pitch: "
- uint8_t Pitch_Buffer [50] = {0}

5.20.1 Detailed Description

Implements the methods from the Display class.

Date

Jun 9, 2024

Author

Jared Sinasohn

Definition in file display_driver.c.

5.20.2 Function Documentation

5.20.2.1 display_note()

```
uint8_t display_note (
            Display * disp,
            uint8_t note )
```

This function displays the current note on the display.

Parameters

disp	The display instance
note	The note to be displayed from 0-11.

Returns

curr_note The note that is displayed.

Attention

The Current function uses uart to display the current note, but at the end would use i2c to the driver.

Definition at line 28 of file display_driver.c.

```
00028                                                               {
00029     if(disp->curr_note == note){
00030         return disp->curr_note;
00031     }
00032     disp->curr_note = note;
00033     switch(disp->curr_note){
00034             case 0:
00035                 sprintf(Pitch_Buffer, "A  \r\n");
00036                 break;
00037             case 1:
00038                 sprintf(Pitch_Buffer, "Bb \r\n");
00039                 break;
00040             case 2:
00041                 sprintf(Pitch_Buffer, "B  \r\n");
00042                 break;
00043             case 3:
00044                 sprintf(Pitch_Buffer, "C  \r\n");
00045                 break;
00046             case 4:
00047                 sprintf(Pitch_Buffer, "Db \r\n");
00048                 break;
00049             case 5:
00050                 sprintf(Pitch_Buffer, "D  \r\n");
00051                 break;
00052             case 6:
00053                 sprintf(Pitch_Buffer, "Eb \r\n");
00054                 break;
00055             case 7:
00056                 sprintf(Pitch_Buffer, "E  \r\n");
00057                 break;
00058             case 8:
00059                 sprintf(Pitch_Buffer, "F  \r\n");
00060                 break;
00061             case 9:
00062                 sprintf(Pitch_Buffer, "Gb \r\n");
00063                 break;
00064             case 10:
00065                 sprintf(Pitch_Buffer, "G  \r\n");
00066                 break;
00067             case 11:
00068                 sprintf(Pitch_Buffer, "Ab \r\n");
00069                 break;
00070         }
00071         HAL_UART_Transmit(disp->huart, Pitch_Message, sizeof(Pitch_Message), 10000);
00072         HAL_UART_Transmit(disp->huart, Pitch_Buffer, sizeof(Pitch_Buffer), 10000);
00073     return disp->curr_note;
00074 }
```

5.20.3 Variable Documentation

5.20.3.1 disp_addr

```
disp_addr = 0b11000001
```

The address of the display driver on the i2c bus

Definition at line 18 of file display_driver.c.

5.20.3.2 note_addresses

```
note_addresses = {0b1101100001111000,0b1101101101110101,0b1101101101110100,0b1101101100000000,0b111000111100l0
```

The values used to light up the 16 segment display, each position in the list corresponds to a note, each bit in the 16 bit numbers corresponds to a segment in the display

Definition at line 13 of file display_driver.c.

```
00013
      {0b1101100001111000,0b1101101101110101,0b1101101101110100,0b1101101100000000,0b1110001111001001,0b1110001111001000,0b11
```

5.20.3.3 Pitch_Buffer

```
uint8_t Pitch_Buffer[50] = {0}
```

Definition at line 20 of file display_driver.c.

```
00020 {0};
```

5.20.3.4 Pitch_Message

```
uint8_t Pitch_Message[] = "Current Pitch:  "
```

Definition at line 19 of file display_driver.c.

5.21 display_driver.c

Go to the documentation of this file.

```
00001
00008 #include "display_driver.h"
00013 uint16_t note_addresses[12] =
      {0b1101100001111000,0b1101101101110101,0b1101101101110100,0b1101101100000000,0b1110001111001001,0b1110001111001000,0b11
00018 uint16_t disp_addr = 0b11000001;
00019 uint8_t Pitch_Message[] = "Current Pitch: ";
00020 uint8_t Pitch_Buffer[50] = {0};
00021
00028 uint8_t display_note(Display* disp, uint8_t note){
00029     if(disp->curr_note == note){
00030         return disp->curr_note;
00031     }
00032     disp->curr_note = note;
00033     switch(disp->curr_note){
00034             case 0:
00035                 sprintf(Pitch_Buffer, "A  \r\n");
00036                 break;
00037             case 1:
00038                 sprintf(Pitch_Buffer, "Bb \r\n");
00039                 break;
00040             case 2:
00041                 sprintf(Pitch_Buffer, "B  \r\n");
00042                 break;
00043             case 3:
00044                 sprintf(Pitch_Buffer, "C  \r\n");
00045                 break;
00046             case 4:
00047                 sprintf(Pitch_Buffer, "Db \r\n");
00048                 break;
00049             case 5:
00050                 sprintf(Pitch_Buffer, "D  \r\n");
00051                 break;
00052             case 6:
00053                 sprintf(Pitch_Buffer, "Eb \r\n");
00054                 break;
00055             case 7:
00056                 sprintf(Pitch_Buffer, "E  \r\n");
00057                 break;
00058             case 8:
```

```
                sprintf(Pitch_Buffer, "F  \r\n");
                break;
            case 9:
                sprintf(Pitch_Buffer, "Gb \r\n");
                break;
            case 10:
                sprintf(Pitch_Buffer, "G  \r\n");
                break;
            case 11:
                sprintf(Pitch_Buffer, "Ab \r\n");
                break;
        }
        HAL_UART_Transmit(disp->huart, Pitch_Message, sizeof(Pitch_Message), 10000);
        HAL_UART_Transmit(disp->huart, Pitch_Buffer, sizeof(Pitch_Buffer), 10000);
    return disp->curr_note;
}

```

5.22 doxy_core/Src/encoder_handler.c File Reference

This file implements the methods in the Encoder class.

```
#include "encoder_handler.h"
```

Functions

- void encoder_read_curr_state (Encoder ∗encoder)
- int32_t encoder_calc_speed (Encoder ∗encoder, int32_t dx, int32_t dt)
- void zero (Encoder ∗encoder)
- int32_t delta (TIM_HandleTypeDef ∗timer, uint32_t initial, uint32_t final)

5.22.1 Detailed Description

This file implements the methods in the Encoder class.

Date

May 23, 2024

Author

Jared Sinasohn

Definition in file encoder_handler.c.

5.22.2 Function Documentation

5.22.2.1 delta()

```
int32_t delta (
            TIM_HandleTypeDef * timer,
            uint32_t initial,
            uint32_t final )
```

This function calculates a delta between two values from a timer, while accounting for overflow.

Parameters

timer	The timer the initial and final values come from
initial	The initial value of the timer
final	The final value of the timer

Returns

delta The corrected delta value to account for overflow and underflow.

get the auto reload value of the timer since it is the maximum value a timer can be

we can determine if something has overflowed or underflowed be assuming a delta will never be greater than half the auto reload value, which if the encoder is read enough is a good assumption

if the value underflows, the delta will be a positive value greater than overflow, so just subtract off ARR+1 from the underflowed delta

similarly, if the value overflows, the delta will be a negative value less than negative of overerflow, so add ARR+1 to the overflowed delta

Definition at line 71 of file encoder_handler.c.

```
00071                                                                    {
00073     uint32_t ARR = (int32_t)(timer->Init.Period );
00075     int32_t overflow = ((ARR-1)/2)+1;
00076     //1 Calculate the delta
00077     int32_t delta = final-initial;
00079     if(delta >= overflow){
00080         delta = delta - overflow*2;
00082     }else if(delta <= -1*overflow){
00083         delta = delta + overflow;
00084     }
00085     return delta;
00086 }
```

5.22.2.2 encoder_calc_speed()

```
int32_t encoder_calc_speed (
              Encoder * encoder,
              int32_t dx,
              int32_t dt )
```

This function calculates a speed based on a differential position and a differential time.

Parameters

encoder	The encoder instance to operate on
dx	The differential position term in encoder counts.
dt	The differential time term in microseconds

Returns

The calculated speed in counts per second

if the delta time is zero, return the previous speed to avoid divide by zero errors.

return the speed in counts/second knowing that dt is in microseconds

Definition at line 39 of file encoder_handler.c.

```
00039                                                                          {
00041     if(dt == 0){
00042         return encoder->speed;
00043     }
00045     return ((dx)*1000000)/dt;
00046 }
```

5.22.2.3 encoder_read_curr_state()

```
void encoder_read_curr_state (
            Encoder * encoder )
```

This function reads the current encoder state and uses it to calculate the speed and position of the encoder.

Parameters

encoder	The encoder instance to operate on.

Attention

If the timing_timer is set up correctly for the inputted encoder, this function should be called no faster than every microsecond.

set the previous times and previous counts to the previously current values

get the count and time values from the two timers

calculate the difference between the current counts/times and previous counts/times using the delta function

set the encoder position to be the previous encoder position plus the delta position

set the speed of the encoder to the calculated value via encoder_calc_speed()

Definition at line 15 of file encoder_handler.c.

```
00015                                                    {
00017     encoder->prev_count = encoder->curr_count;
00018     encoder->prev_time = encoder->curr_time;
00020     encoder->curr_count = encoder->timer->Instance->CNT;
00021     encoder->curr_time = encoder->timing_timer->Instance->CNT;
00023     encoder->dx = delta(encoder->timer, encoder->prev_count,encoder->curr_count);
00024     encoder->dt = delta(encoder->timing_timer, encoder->prev_time,encoder->curr_time);
00026     encoder->pos = encoder->pos + encoder->dx;
00028     encoder->speed = encoder_calc_speed(encoder,encoder->dx,encoder->dt);
00029 
00030 }
```

5.22.2.4 zero()

```
void zero (
            Encoder * encoder )
```

This function zeros the encoder, including the timing_timer and timer timers.

Parameters

encoder	The encoder instance to operate on

Definition at line 52 of file encoder_handler.c.

```
00052                             {
00053     encoder->timer->Instance->CNT = 0;
00054     encoder->timing_timer->Instance->CNT = 0;
00055     encoder->prev_count = 0;
00056     encoder->curr_count = 0;
00057     encoder->prev_time = 0;
00058     encoder->curr_time = 1;
00059     encoder->pos = 0;
00060     encoder->speed = 0;
00061 }
```

5.23 encoder_handler.c

Go to the documentation of this file.

```
00001
00008 #include "encoder_handler.h"
00009
00015 void encoder_read_curr_state(Encoder* encoder){
00017     encoder->prev_count = encoder->curr_count;
00018     encoder->prev_time = encoder->curr_time;
00020     encoder->curr_count = encoder->timer->Instance->CNT;
00021     encoder->curr_time = encoder->timing_timer->Instance->CNT;
00023     encoder->dx = delta(encoder->timer, encoder->prev_count,encoder->curr_count);
00024     encoder->dt = delta(encoder->timing_timer, encoder->prev_time,encoder->curr_time);
00026     encoder->pos = encoder->pos + encoder->dx;
00028     encoder->speed = encoder_calc_speed(encoder,encoder->dx,encoder->dt);
00029
00030 }
00031
00039 int32_t encoder_calc_speed(Encoder* encoder, int32_t dx,int32_t dt){
00041     if(dt == 0){
00042         return encoder->speed;
00043     }
00045     return ((dx)*1000000)/dt;
00046 }
00047
00052 void zero(Encoder* encoder){
00053     encoder->timer->Instance->CNT = 0;
00054     encoder->timing_timer->Instance->CNT = 0;
00055     encoder->prev_count = 0;
00056     encoder->curr_count = 0;
00057     encoder->prev_time = 0;
00058     encoder->curr_time = 1;
00059     encoder->pos = 0;
00060     encoder->speed = 0;
00061 }
00062
00071 int32_t delta(TIM_HandleTypeDef* timer, uint32_t initial, uint32_t final){
00073     uint32_t ARR = (int32_t)(timer->Init.Period );
00075     int32_t overflow = ((ARR-1)/2)+1;
00076     //1 Calculate the delta
00077     int32_t delta = final-initial;
00079     if(delta >= overflow){
00080         delta = delta - overflow*2;
00082     }else if(delta <= -1*overflow){
00083         delta = delta + overflow;
00084     }
00085     return delta;
00086 }
00087
00088
00089
```

5.24 doxy_core/Src/main.c File Reference

: Main program body

```
#include "main.h"
#include "stm32l4xx_hal.h"
#include "CLController.h"
#include "controller_driver.h"
#include "display_driver.h"
#include "encoder_handler.h"
#include "motor_driver.h"
#include "pitch_encoder_handler.h"
#include <stdio.h>
```

Functions

- void SystemClock_Config (void)

  System Clock Configuration.
- void display_task (uint8_t ∗state)
- void motor_task (uint8_t ∗state)
- int main (void)

  The application entry point.
- void Error_Handler (void)

  This function is executed in case of error occurrence.

Variables

- ADC_HandleTypeDef hadc1
- I2C_HandleTypeDef hi2c2
- TIM_HandleTypeDef htim1
- TIM_HandleTypeDef htim3
- TIM_HandleTypeDef htim4
- TIM_HandleTypeDef htim5
- TIM_HandleTypeDef htim6
- TIM_HandleTypeDef htim8
- UART_HandleTypeDef huart2
- uint8_t t1state = 0
- uint8_t t2state = 0
- uint8_t Buffer [50] = {0}

  These are buffers used for displaying stuff via uart during debugging.
- uint8_t Pos_Buffer [50] = {0}
- uint8_t Speed_Buffer [50] = {0}
- uint8_t Space [ ] = " - "
- uint8_t StartMSG [ ] = "Starting I2C Scanning: \r\n"
- uint8_t EndMSG [ ] = "Done! \r\n\r\n"
- uint8_t led_buff
- uint8_t Eff_Buffer [50] = {0}

5.24.1 Detailed Description

: Main program body

!

Attention

Definition in file main.c.

5.24.2 Function Documentation

5.24.2.1 display_task()

```
void display_task (
            uint8_t * state )
```

This function implements the Display and Note task finite state machine.

Parameters

state	The pointer to the current state of the task

Definition at line 233 of file main.c.

```
00233                                 {
00234     switch(*state){
00235     case 0:
00236         pe.encoder->timer->Instance->CNT = ((htim5.Init.Period)+1)/2;
00237         *state = 1;
00238         break;
00239     case 1:
00240         ptch = get_pitch(&pe);
00241         display_note(&display,ptch);
00242         break;
00243     }
00244 }
```

5.24.2.2 Error_Handler()

```
void Error_Handler (
            void )
```

This function is executed in case of error occurrence.

Return values

None	

Definition at line 1054 of file main.c.

```
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}
```

5.24.2.3 main()

```
int main (
            void )
```

The application entry point.

Return values

int	

initialize I2C

start the timers in their various modes.

set the display driver enable pin and the motor driver enable pin

This code was all used to debug the i2c driver to no avail, I have left it here for transparancy.

This is the game loop that runs forever to convert user inputed code into a PWM signal to drive the motors.

run the tasks in round-robin style.

Definition at line 273 of file main.c.

```
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM1_Init();
  MX_TIM3_Init();
  MX_ADC1_Init();
  MX_I2C2_Init();
  MX_TIM4_Init();
  MX_TIM5_Init();
  MX_TIM8_Init();
  MX_TIM6_Init();
  /* USER CODE BEGIN 2 */
  MX_I2C2_Init();
```

```
HAL_TIM_Encoder_Start(&htim4, TIM_CHANNEL_ALL);
HAL_TIM_Encoder_Start_IT(&htim5, TIM_CHANNEL_ALL);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
HAL_TIM_Base_Start(&htim6);
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10,GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2,GPIO_PIN_SET);
/*uint8_t aTxBuffer[2] = {};
aTxBuffer[0] = 0x00;
aTxBuffer[1] = 0x00;
uint8_t lvl = 85;
uint8_t addr = 0b11000000;
HAL_I2C_Mem_Read(&hi2c2, addr, 0x00, 1, led_buff, 1, 500);
  sprintf(Buffer, "Reg 0x14x: %X \r\n",led_buff);
  HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
HAL_Delay(500);
ret = HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x14, 1, lvl, 1, 500);
/*HAL_Delay(500);
if(ret!=HAL_OK){
    sprintf(Buffer, "did not work :(\n\r");
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
}else if(ret == HAL_OK)
{
    sprintf(Buffer, "worked!\n\r");
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
}
HAL_Delay(500);
ret = HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x15, 1, lvl, 1, 500);
  /*if(ret!=HAL_OK){
    sprintf(Buffer, "did not work :(\n\r");
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }else if(ret == HAL_OK)
  {
      sprintf(Buffer, "worked!\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }
HAL_Delay(500);
ret = HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x16, 1, lvl, 1, 500);
  /*if(ret!=HAL_OK){
    sprintf(Buffer, "did not work :(\n\r");
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }else if(ret == HAL_OK)
  {
      sprintf(Buffer, "worked!\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }
HAL_Delay(500);
ret = HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x17, 1, lvl, 1, 500);
  /*if(ret!=HAL_OK){
    sprintf(Buffer, "did not work :(\n\r");
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }else if(ret == HAL_OK)
  {
      sprintf(Buffer, "worked!\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }
HAL_I2C_Mem_Read(&hi2c2, addr, 0x14, 1, (uint8_t*)led_buff, 1, 500);
sprintf(Buffer, "Reg 0x14x: %X \r\n",led_buff);
HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
HAL_I2C_Mem_Read(&hi2c2, addr, 0x15, 1, led_buff, 1, 500);
sprintf(Buffer, "Reg 0x15x: %X \r\n",led_buff);
  HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
HAL_I2C_Mem_Read(&hi2c2, addr, 0x16, 1, led_buff, 1, 500);
sprintf(Buffer, "Reg 0x16x: %X \r\n",led_buff);
  HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
HAL_I2C_Mem_Read(&hi2c2, addr, 0x17, 1, led_buff, 1, 500);
sprintf(Buffer, "Reg 0x17x: %X \r\n",led_buff);
  HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
/*if(ret!=HAL_OK){
    sprintf(Buffer, "did not work :(\n\r");
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }else if(ret == HAL_OK)
  {
      sprintf(Buffer, "worked!\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }*/
//HAL_Delay(500);
//ret = HAL_I2C_Master_Transmit(&hi2c2, 0b11000001, ((uint8_t*)1),1,100);
/*if(ret!=HAL_OK){
    sprintf(Buffer, "did not work :(\n\r");
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }else if(ret == HAL_OK)
  {
      sprintf(Buffer, "worked!\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }
      uint32_t time1 = HAL_GetTick();
```

```
        uint32_t time2 = HAL_GetTick();
        int32_t sped = 0;
        //HAL_I2C_Master_Transmit((&hi2c2), 0b11000001, ((uint8_t*)0b1101100001111000),2,100);
        uint8_t prev_count = 1;

  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x02, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x03, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x04, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x05, 1, (uint8_t*)0xFF, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x06, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x07, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x08, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x09, 1, (uint8_t*)0xFF, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0A, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0B, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0C, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0D, 1, (uint8_t*)0xFF, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0E, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0F, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x10, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x11, 1, (uint8_t*)0xFF, 1, 100);
   // The folloqing code was taken from here:
https://deepbluembedded.com/stm32-i2c-scanner-hal-code-example/
  uint8_t TLC59116_PWM0_AUTOINCR = 0x82;
  HAL_UART_Transmit(&huart2, StartMSG, sizeof(StartMSG), 10000);
      for(i=1; i<128; i++)
      {
          ret = HAL_I2C_IsDeviceReady(&hi2c2, (uint16_t)(i«1), 3, 5);
          if (ret != HAL_OK)
          {
              HAL_UART_Transmit(&huart2, Space, sizeof(Space), 10000);
          }
          else if(ret == HAL_OK)
          {
              sprintf(Buffer, "0x%X", i);
              HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
          }
      }
      HAL_UART_Transmit(&huart2, EndMSG, sizeof(EndMSG), 10000);
  uint8_t bruh[] = { TLC59116_PWM0_AUTOINCR, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
  while(HAL_I2C_Master_Transmit(&hi2c2, 0xC0, *bruh, sizeof(bruh), 100) != HAL_OK)
      {
      HAL_Delay(1);
      if (HAL_I2C_GetError(&hi2c2) != HAL_I2C_ERROR_AF)
          {
              Error_Handler();
          }
      }*/
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    display_task(&t1state);
    motor_task(&t2state);

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

5.24.2.4 motor_task()

```
void motor_task (
        uint8_t * state )
```

This function implements the Motor task finite state machine.

Parameters

state	The pointer to the current state of the task

Definition at line 250 of file main.c.

```
                              {
    switch(*state){
        case 0:
            *state = 1;
            timmy = HAL_GetTick();
            break;
        case 1:
            if(HAL_GetTick() >= timmy + 2){
                timmy = HAL_GetTick();
                m_con.setpoint = motor_speeds[ptch];
                encoder_read_curr_state(&mot_enc);
                eff = run(&m_con,mot_enc.speed);
                motor_set_duty_cycle(&m, eff);
                break;
            }
    }
}
```

5.24.2.5 SystemClock_Config()

```
void SystemClock_Config (
              void   )
```

System Clock Configuration.

Return values

None	

Configure the main internal regulator output voltage

Initializes the RCC Oscillators according to the specified parameters in the RCC_OscInitTypeDef structure.

Initializes the CPU, AHB and APB buses clocks

Definition at line 469 of file main.c.

```
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};


  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
  {
    Error_Handler();
  }


  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 1;
  RCC_OscInitStruct.PLL.PLLN = 10;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }


  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
  {
    Error_Handler();
  }
}
```

5.24.3 Variable Documentation

5.24.3.1 Buffer

```
uint8_t Buffer[50] = {0}
```

These are buffers used for displaying stuff via uart during debugging.

Definition at line 176 of file main.c.

```
00176 {0};
```

5.24.3.2 Eff_Buffer

```
uint8_t Eff_Buffer[50] = {0}
```

Definition at line 183 of file main.c.

```
00183 {0};
```

5.24.3.3 EndMSG

```
uint8_t EndMSG[] = "Done!  \r\n\r\n"
```

Definition at line 181 of file main.c.

5.24.3.4 hadc1

```
ADC_HandleTypeDef hadc1
```

Definition at line 50 of file main.c.

5.24.3.5 hi2c2

```
I2C_HandleTypeDef hi2c2
```

Definition at line 52 of file main.c.

5.24.3.6 htim1

```
TIM_HandleTypeDef htim1
```

Definition at line 54 of file main.c.

5.24.3.7 htim3

```
TIM_HandleTypeDef htim3
```

Definition at line 55 of file main.c.

5.24.3.8 htim4

```
TIM_HandleTypeDef htim4
```

Definition at line 56 of file main.c.

5.24.3.9 htim5

```
TIM_HandleTypeDef htim5
```

Definition at line 57 of file main.c.

5.24.3.10 htim6

```
TIM_HandleTypeDef htim6
```

Definition at line 58 of file main.c.

5.24.3.11 htim8

```
TIM_HandleTypeDef htim8
```

Definition at line 59 of file main.c.

5.24.3.12 huart2

```
UART_HandleTypeDef huart2
```

Definition at line 61 of file main.c.

5.24.3.13 led_buff

```
uint8_t led_buff
```

Definition at line 182 of file main.c.

5.24.3.14 Pos_Buffer

```
uint8_t Pos_Buffer[50] = {0}
```

Definition at line 177 of file main.c.

```
00177 {0};
```

5.24.3.15 Space

```
uint8_t Space[] = " - "
```

Definition at line 179 of file main.c.

5.24.3.16 Speed_Buffer

```
uint8_t Speed_Buffer[50] = {0}
```

Definition at line 178 of file main.c.

```
00178 {0};
```

5.24.3.17 StartMSG

```
uint8_t StartMSG[] = "Starting I2C Scanning:  \r\n"
```

Definition at line 180 of file main.c.

5.24.3.18 t1state

```
t1state = 0
```

The current state of the Display and note task

Definition at line 167 of file main.c.

5.24.3.19 t2state

```
t2state = 0
```

The current state of the motor task

Definition at line 173 of file main.c.

5.25 main.c

Go to the documentation of this file.

```
/* USER CODE BEGIN Header */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "stm32l4xx_hal.h"
#include "CLController.h"
#include "controller_driver.h"
#include "display_driver.h"
#include "encoder_handler.h"
#include "motor_driver.h"
#include "pitch_encoder_handler.h"
#include <stdio.h>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;

I2C_HandleTypeDef hi2c2;

TIM_HandleTypeDef htim1;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;
TIM_HandleTypeDef htim5;
TIM_HandleTypeDef htim6;
TIM_HandleTypeDef htim8;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */
static RC_Controller con = {.timer = &htim3,
          .timer = &htim3,
          .period1 = 1500,
          .period2 = 1500,
          .re1 = 0,
          .re2 = 0,
          .fe1 = 0,
          .fe2 = 0,
          .fe_flag1 = 0,
          .fe_flag2 = 0,
              };

static Motor m = { .timer = &htim1,
           .channels = 1,
           .duty_cycle = 0,
         .enable_flag = 1
          };

static Encoder pitch = {.timer = &htim5,
                         .timing_timer = &htim6,
                         .prev_count = 0,
                         .curr_count = 0,
                         .prev_time = 0,
                         .curr_time = 1,
                         .pos = 0,
                         .speed = 0,
                         .dx = 0,
                         .dt = 0
};

static Encoder mot_enc = {.timer = &htim4,
                         .timing_timer = &htim6,
                         .prev_count = 0,
                         .curr_count = 0,
                         .prev_time = 0,
```

```c
                              .curr_time = 1,
                              .pos = 0,
                              .speed = 0,
                              .dx = 0,
                              .dt = 0
};

static CLController m_con = {
        .kp = 500,
        .ki = 0,
        .kd = 0,
        .kf = 706,
        .setpoint = 0,
        .eff = 0,
        .curr = 0,
        .err = 0,
        .err_acc = 0,
        .prev_err_index = 0,
        .initial_time = 0,
        .curr_time = 1,
        .slope = 0,
        .prev_err_list_length = 10,
        .prev_err_list = {0,0,0,0,0,0,0,0,0,0}
};

static PitchEncoder pe = {.pitch = 0,
                                .encoder = &pitch,
                                .delta = 0
};

static Display display = {.hi2c = &hi2c2,
                            .curr_note = 0,
                            .huart = &huart2
};

uint8_t t1state = 0;

uint8_t t2state = 0;

uint8_t Buffer[50] = {0};
uint8_t Pos_Buffer[50] = {0};
uint8_t Speed_Buffer[50] = {0};
uint8_t Space[] = " - ";
uint8_t StartMSG[] = "Starting I2C Scanning: \r\n";
uint8_t EndMSG[] = "Done! \r\n\r\n";
uint8_t led_buff;
uint8_t Eff_Buffer[50] = {0};

static int32_t motor_speeds[12] = {56320,59679,63222,66970,70963,75182,79647,84378,89395,94720,100352,106312};

static uint32_t ptch;

static uint32_t timmy = 0;

static int32_t eff = 0;
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_TIM1_Init(void);
static void MX_TIM3_Init(void);
static void MX_ADC1_Init(void);
static void MX_I2C2_Init(void);
static void MX_TIM4_Init(void);
static void MX_TIM5_Init(void);
static void MX_TIM8_Init(void);
static void MX_TIM6_Init(void);
/* USER CODE BEGIN PFP */
void display_task(uint8_t* state);
void motor_task (uint8_t* state);
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void display_task(uint8_t* state){
    switch(*state){
    case 0:
        pe.encoder->timer->Instance->CNT = ((htim5.Init.Period)+1)/2;
        *state = 1;
        break;
    case 1:
        ptch = get_pitch(&pe);
        display_note(&display,ptch);
        break;
```

```
    }
}

void motor_task (uint8_t* state){
    switch(*state){
        case 0:
            *state = 1;
            timmy = HAL_GetTick();
            break;
        case 1:
            if(HAL_GetTick() >= timmy + 2){
                timmy = HAL_GetTick();
                m_con.setpoint = motor_speeds[ptch];
                encoder_read_curr_state(&mot_enc);
                eff = run(&m_con,mot_enc.speed);
                motor_set_duty_cycle(&m, eff);
                break;
            }
    }
}
/* USER CODE END 0 */

int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM1_Init();
  MX_TIM3_Init();
  MX_ADC1_Init();
  MX_I2C2_Init();
  MX_TIM4_Init();
  MX_TIM5_Init();
  MX_TIM8_Init();
  MX_TIM6_Init();
  /* USER CODE BEGIN 2 */
  MX_I2C2_Init();
  HAL_TIM_Encoder_Start(&htim4, TIM_CHANNEL_ALL);
  HAL_TIM_Encoder_Start_IT(&htim5, TIM_CHANNEL_ALL);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
  HAL_TIM_Base_Start(&htim6);
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10,GPIO_PIN_SET);
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2,GPIO_PIN_SET);
  /*uint8_t aTxBuffer[2] = {};
  aTxBuffer[0] = 0x00;
  aTxBuffer[1] = 0x00;
  uint8_t lvl = 85;
  uint8_t addr = 0b11000000;
  HAL_I2C_Mem_Read(&hi2c2, addr, 0x00, 1, led_buff, 1, 500);
    sprintf(Buffer, "Reg 0x14x: %X \r\n",led_buff);
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  HAL_Delay(500);
  ret = HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x14, 1, lvl, 1, 500);
  /*HAL_Delay(500);
  if(ret!=HAL_OK){
      sprintf(Buffer, "did not work :(\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }else if(ret == HAL_OK)
  {
      sprintf(Buffer, "worked!\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  }
  HAL_Delay(500);
  ret = HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x15, 1, lvl, 1, 500);
    /*if(ret!=HAL_OK){
```

```
      sprintf(Buffer, "did not work :(\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }else if(ret == HAL_OK)
    {
        sprintf(Buffer, "worked!\n\r");
        HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }
  HAL_Delay(500);
  ret = HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x16, 1, lvl, 1, 500);
    /*if(ret!=HAL_OK){
      sprintf(Buffer, "did not work :(\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }else if(ret == HAL_OK)
    {
        sprintf(Buffer, "worked!\n\r");
        HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }
  HAL_Delay(500);
  ret = HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x17, 1, lvl, 1, 500);
    /*if(ret!=HAL_OK){
      sprintf(Buffer, "did not work :(\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }else if(ret == HAL_OK)
    {
        sprintf(Buffer, "worked!\n\r");
        HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }
  HAL_I2C_Mem_Read(&hi2c2, addr, 0x14, 1, (uint8_t*)led_buff, 1, 500);
  sprintf(Buffer, "Reg 0x14x: %X \r\n",led_buff);
  HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  HAL_I2C_Mem_Read(&hi2c2, addr, 0x15, 1, led_buff, 1, 500);
  sprintf(Buffer, "Reg 0x15x: %X \r\n",led_buff);
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  HAL_I2C_Mem_Read(&hi2c2, addr, 0x16, 1, led_buff, 1, 500);
  sprintf(Buffer, "Reg 0x16x: %X \r\n",led_buff);
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  HAL_I2C_Mem_Read(&hi2c2, addr, 0x17, 1, led_buff, 1, 500);
  sprintf(Buffer, "Reg 0x17x: %X \r\n",led_buff);
    HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
  /*if(ret!=HAL_OK){
      sprintf(Buffer, "did not work :(\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }else if(ret == HAL_OK)
    {
        sprintf(Buffer, "worked!\n\r");
        HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }*/
  //HAL_Delay(500);
  //ret = HAL_I2C_Master_Transmit(&hi2c2, 0b11000001, ((uint8_t*)1),1,100);
  /*if(ret!=HAL_OK){
      sprintf(Buffer, "did not work :(\n\r");
      HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }else if(ret == HAL_OK)
    {
        sprintf(Buffer, "worked!\n\r");
        HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
    }
        uint32_t time1 = HAL_GetTick();
        uint32_t time2 = HAL_GetTick();
        int32_t sped = 0;
        //HAL_I2C_Master_Transmit((&hi2c2), 0b11000001, ((uint8_t*)0b1101100001111000),2,100);
        uint8_t prev_count = 1;

  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x02, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x03, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x04, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x05, 1, (uint8_t*)0xFF, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x06, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x07, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x08, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x09, 1, (uint8_t*)0xFF, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0A, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0B, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0C, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0D, 1, (uint8_t*)0xFF, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0E, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x0F, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x10, 1, (uint8_t*)lvl, 1, 100);
  //HAL_I2C_Mem_Write(&hi2c2, 0b11000000, 0x11, 1, (uint8_t*)0xFF, 1, 100);
    // The folloqing code was taken from here: https://deepbluembedded.com/stm32-i2c-scanner-hal-code-example/
  uint8_t TLC59116_PWM0_AUTOINCR = 0x82;
  HAL_UART_Transmit(&huart2, StartMSG, sizeof(StartMSG), 10000);
      for(i=1; i<128; i++)
      {
          ret = HAL_I2C_IsDeviceReady(&hi2c2, (uint16_t)(i<<1), 3, 5);
          if (ret != HAL_OK)
```

```
          {
              HAL_UART_Transmit(&huart2, Space, sizeof(Space), 10000);
          }
          else if(ret == HAL_OK)
          {
              sprintf(Buffer, "0x%X", i);
              HAL_UART_Transmit(&huart2, Buffer, sizeof(Buffer), 10000);
          }
      }
      HAL_UART_Transmit(&huart2, EndMSG, sizeof(EndMSG), 10000);
  uint8_t bruh[] = { TLC59116_PWM0_AUTOINCR, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
  while(HAL_I2C_Master_Transmit(&hi2c2, 0xC0, *bruh, sizeof(bruh), 100) != HAL_OK)
      {
      HAL_Delay(1);
      if (HAL_I2C_GetError(&hi2c2) != HAL_I2C_ERROR_AF)
          {
              Error_Handler();
          }
      }*/
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    display_task(&t1state);
    motor_task(&t2state);

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
  {
    Error_Handler();
  }

  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 1;
  RCC_OscInitStruct.PLL.PLLN = 10;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
  {
    Error_Handler();
  }
}

static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_MultiModeTypeDef multimode = {0};
  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */

```

```c
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc1.Init.LowPowerAutoWait = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  hadc1.Init.OversamplingMode = DISABLE;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }

  multimode.Mode = ADC_MODE_INDEPENDENT;
  if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
  {
    Error_Handler();
  }

  sConfig.Channel = ADC_CHANNEL_14;
  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
  sConfig.SingleDiff = ADC_SINGLE_ENDED;
  sConfig.OffsetNumber = ADC_OFFSET_NONE;
  sConfig.Offset = 0;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

static void MX_I2C2_Init(void)
{

  /* USER CODE BEGIN I2C2_Init 0 */

  /* USER CODE END I2C2_Init 0 */

  /* USER CODE BEGIN I2C2_Init 1 */

  /* USER CODE END I2C2_Init 1 */
  hi2c2.Instance = I2C2;
  hi2c2.Init.Timing = 0xF010F3FE;
  hi2c2.Init.OwnAddress1 = 0;
  hi2c2.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c2.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c2.Init.OwnAddress2 = 0;
  hi2c2.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
  hi2c2.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c2.Init.NoStretchMode = I2C_NOSTRETCH_ENABLE;
  if (HAL_I2C_Init(&hi2c2) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_I2CEx_ConfigAnalogFilter(&hi2c2, I2C_ANALOGFILTER_ENABLE) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_I2CEx_ConfigDigitalFilter(&hi2c2, 0) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN I2C2_Init 2 */

  /* USER CODE END I2C2_Init 2 */

}

static void MX_TIM1_Init(void)
{

  /* USER CODE BEGIN TIM1_Init 0 */

  /* USER CODE END TIM1_Init 0 */
```

```

  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};
  TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = {0};

  /* USER CODE BEGIN TIM1_Init 1 */

  /* USER CODE END TIM1_Init 1 */
  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 0;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 999;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_PWM_Init(&htim1) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
  sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
  if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
  {
    Error_Handler();
  }
  sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;
  sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
  sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
  sBreakDeadTimeConfig.DeadTime = 0;
  sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
  sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
  sBreakDeadTimeConfig.BreakFilter = 0;
  sBreakDeadTimeConfig.Break2State = TIM_BREAK2_DISABLE;
  sBreakDeadTimeConfig.Break2Polarity = TIM_BREAK2POLARITY_HIGH;
  sBreakDeadTimeConfig.Break2Filter = 0;
  sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
  if (HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM1_Init 2 */

  /* USER CODE END TIM1_Init 2 */
  HAL_TIM_MspPostInit(&htim1);

}

static void MX_TIM3_Init(void)
{

  /* USER CODE BEGIN TIM3_Init 0 */

  /* USER CODE END TIM3_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_IC_InitTypeDef sConfigIC = {0};

  /* USER CODE BEGIN TIM3_Init 1 */

  /* USER CODE END TIM3_Init 1 */
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 79;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 65535;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_IC_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
```

```
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
  sConfigIC.ICSelection = TIM_ICSELECTION_INDIRECTTI;
  sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
  sConfigIC.ICFilter = 0;
  if (HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
  sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
  if (HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_2) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
  if (HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_3) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
  sConfigIC.ICSelection = TIM_ICSELECTION_INDIRECTTI;
  if (HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_4) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM3_Init 2 */

  /* USER CODE END TIM3_Init 2 */

}

static void MX_TIM4_Init(void)
{

  /* USER CODE BEGIN TIM4_Init 0 */

  /* USER CODE END TIM4_Init 0 */

  TIM_Encoder_InitTypeDef sConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM4_Init 1 */

  /* USER CODE END TIM4_Init 1 */
  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 0;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 65535;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  sConfig.EncoderMode = TIM_ENCODERMODE_TI12;
  sConfig.IC1Polarity = TIM_ICPOLARITY_RISING;
  sConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
  sConfig.IC1Prescaler = TIM_ICPSC_DIV1;
  sConfig.IC1Filter = 0;
  sConfig.IC2Polarity = TIM_ICPOLARITY_RISING;
  sConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
  sConfig.IC2Prescaler = TIM_ICPSC_DIV1;
  sConfig.IC2Filter = 0;
  if (HAL_TIM_Encoder_Init(&htim4, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM4_Init 2 */

  /* USER CODE END TIM4_Init 2 */

}

static void MX_TIM5_Init(void)
{

  /* USER CODE BEGIN TIM5_Init 0 */

  /* USER CODE END TIM5_Init 0 */

```

```
  TIM_Encoder_InitTypeDef sConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM5_Init 1 */

  /* USER CODE END TIM5_Init 1 */
  htim5.Instance = TIM5;
  htim5.Init.Prescaler = 0;
  htim5.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim5.Init.Period = 1073741823;
  htim5.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim5.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  sConfig.EncoderMode = TIM_ENCODERMODE_TI12;
  sConfig.IC1Polarity = TIM_ICPOLARITY_RISING;
  sConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
  sConfig.IC1Prescaler = TIM_ICPSC_DIV1;
  sConfig.IC1Filter = 0;
  sConfig.IC2Polarity = TIM_ICPOLARITY_RISING;
  sConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
  sConfig.IC2Prescaler = TIM_ICPSC_DIV1;
  sConfig.IC2Filter = 0;
  if (HAL_TIM_Encoder_Init(&htim5, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_OnePulse_Init(&htim5, TIM_OPMODE_SINGLE) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim5, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM5_Init 2 */

  /* USER CODE END TIM5_Init 2 */

}

static void MX_TIM6_Init(void)
{

  /* USER CODE BEGIN TIM6_Init 0 */

  /* USER CODE END TIM6_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM6_Init 1 */

  /* USER CODE END TIM6_Init 1 */
  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 79;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = 65535;
  htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM6_Init 2 */

  /* USER CODE END TIM6_Init 2 */

}

static void MX_TIM8_Init(void)
{

  /* USER CODE BEGIN TIM8_Init 0 */

  /* USER CODE END TIM8_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_IC_InitTypeDef sConfigIC = {0};

  /* USER CODE BEGIN TIM8_Init 1 */

  /* USER CODE END TIM8_Init 1 */
```

```
  htim8.Instance = TIM8;
  htim8.Init.Prescaler = 0;
  htim8.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim8.Init.Period = 65535;
  htim8.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim8.Init.RepetitionCounter = 0;
  htim8.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_IC_Init(&htim8) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim8, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
  sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
  sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
  sConfigIC.ICFilter = 0;
  if (HAL_TIM_IC_ConfigChannel(&htim8, &sConfigIC, TIM_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
  sConfigIC.ICSelection = TIM_ICSELECTION_INDIRECTTI;
  if (HAL_TIM_IC_ConfigChannel(&htim8, &sConfigIC, TIM_CHANNEL_2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM8_Init 2 */

  /* USER CODE END TIM8_Init 2 */

}

static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);

  /*Configure GPIO pin : PB2 */
```

```
  GPIO_InitStruct.Pin = GPIO_PIN_2;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /*Configure GPIO pin : PA10 */
  GPIO_InitStruct.Pin = GPIO_PIN_10;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pin : PC10 */
  GPIO_InitStruct.Pin = GPIO_PIN_10;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

5.26 doxy_core/Src/motor_driver.c File Reference

this file implements the Motor struct to allow for pseudo object oriented programming motor control.

```
#include "motor_driver.h"
```

Functions

- void motor_set_duty_cycle (Motor *motor, int32_t doot)
- void motor_enable_disable (Motor *motor, uint8_t enable)

5.26.1 Detailed Description

this file implements the Motor struct to allow for pseudo object oriented programming motor control.

Date

Apr 25, 2024

Author

Jared Sinasohn

Definition in file motor_driver.c.

5.26.2 Function Documentation

5.26.2.1 motor_enable_disable()

```
void motor_enable_disable (
            Motor * motor,
            uint8_t enable )
```

Enables or disables motor based on user input

Parameters

motor,the	Motor struct to act upon
enable,the	boolean of whether to enable or disable the motor with 1 being to enable and 0 being to disable.

Definition at line 87 of file motor_driver.c.

```
00087                                                         {
00088     // if user wants to enable motor
00089     if(enable == 1){
00090         motor->enable_flag = 1;
00091         // First retrieve ARR to set motor to brake mode
00092         uint32_t ARR = (uint32_t)(motor->timer->Init.Period + 1);
00093 
00094         // Now set the correct motor pair to brake mode.
00095         if(motor->channels == 1){
00096             motor->timer->Instance->CCR1 = ARR;
00097             motor->timer->Instance->CCR2 = ARR;
00098         } else if(motor->channels == 2){
00099             motor->timer->Instance->CCR3 = ARR;
00100             motor->timer->Instance->CCR4 = ARR;
00101         }else{
00102             return;
00103         }
00104 
00105         // set the motor's enable flag to 1
00106         motor->enable_flag = 1;
00107 
00108     // if user wants to disable motor
00109     } else if(enable == 0){
00110         motor->enable_flag = 0;
00111         if(motor->channels == 1){
00112             motor->timer->Instance->CCR1 = 0;
00113             motor->timer->Instance->CCR2 = 0;
00114         } else if(motor->channels == 2){
00115             motor->timer->Instance->CCR3 = 0;
00116             motor->timer->Instance->CCR4 = 0;
00117         }else{
00118             return;
00119         }
00120 
00121         // set the motor's enable flag to 0
00122         motor->enable_flag = 0;
00123     }
00124 }
```

5.26.2.2 motor_set_duty_cycle()

```
void motor_set_duty_cycle (
            Motor * motor,
            int32_t doot )
```

This function implements the duty cycle setting of the motor. It takes in the motor struct and a duty cycle from -100 to 100 (though the function saturates values above and below these values).

Parameters

motor,the	Motor struct to be operated on.
doot,the	duty cycle to be set to.

Definition at line 21 of file motor_driver.c.

```
00021                                                              {
00022     motor->duty_cycle = doot;
00023     // First, check if the motor is disabled
00024     if(motor->enable_flag != 1){
00025         // if the enable flag isn't set exit the function and do nothing.
00026         // we are also using != 1 so if there is a stray value in memory,
00027         // the motor doesn't accidentally enable.
00028         return;
00029     }
00030 
00031     // Next, saturate the duty cycle just in case.
00032     if(doot < -100){
00033         doot = -100;
00034     }
00035     if(doot > 100){
00036         doot = 100;
00037     }
00038 
00039     // We need to get the auto reload value for the timer we are using
00040     // signed value so we don't run into sign issues later
00041     int32_t ARR = (int32_t)(motor->timer->Init.Period + 1);
00042 
00043     // Now calculate the duty cycle in terms of the CCR value
00044     doot = doot*ARR/100;  // multiply first so we don't lose data
00045 
00046     // now we need to set the motors to the correct duty cycles
00047     // Forwards will be channels 1 and 3 for motors 1 and 2 respectively
00048     // Backwards will be channels 2 and 4 for motors 1 and 2 respectively
00049 
00050 
00051     // the below CCR's are based on the logic table of the toshiba, setting motor.
00052     // to brake mode
00053     // if duty cycle is <0
00054     if (doot < 0){
00055         // check if it is the first or second motor.
00056         if(motor->channels == 1){
00057             motor->timer->Instance->CCR1 = ARR;
00058             motor->timer->Instance->CCR2 = ARR + doot;
00059         } else if(motor->channels == 2){
00060             motor->timer->Instance->CCR3 = ARR;
00061             motor->timer->Instance->CCR4 = ARR + doot;
00062         }else{
00063             // if neither return
00064             return;
00065         }
00066     // if duty cycle >=0
00067     } else{
00068         if(motor->channels == 1){
00069             motor->timer->Instance->CCR1 = ARR - doot;
00070             motor->timer->Instance->CCR2 = ARR;
00071         } else if(motor->channels == 2){
00072             motor->timer->Instance->CCR3 = ARR - doot;
00073             motor->timer->Instance->CCR4 = ARR;
00074         }else{
00075             return;
00076         }
00077     }
00078 }
```

5.27 motor_driver.c

Go to the documentation of this file.

```
00001 
00011 #include "motor_driver.h"
00012 
00021 void motor_set_duty_cycle(Motor* motor, int32_t doot){
00022     motor->duty_cycle = doot;
00023     // First, check if the motor is disabled
00024     if(motor->enable_flag != 1){
00025         // if the enable flag isn't set exit the function and do nothing.
00026         // we are also using != 1 so if there is a stray value in memory,
00027         // the motor doesn't accidentally enable.
00028         return;
00029     }
00030 
00031     // Next, saturate the duty cycle just in case.
00032     if(doot < -100){
00033         doot = -100;
00034     }
00035     if(doot > 100){
```

```
        doot = 100;
    }

    // We need to get the auto reload value for the timer we are using
    // signed value so we don't run into sign issues later
    int32_t ARR = (int32_t)(motor->timer->Init.Period + 1);

    // Now calculate the duty cycle in terms of the CCR value
    doot = doot*ARR/100;  // multiply first so we don't lose data

    // now we need to set the motors to the correct duty cycles
    // Forwards will be channels 1 and 3 for motors 1 and 2 respectively
    // Backwards will be channels 2 and 4 for motors 1 and 2 respectively


    // the below CCR's are based on the logic table of the toshiba, setting motor.
    // to brake mode
    // if duty cycle is <0
    if (doot < 0){
        // check if it is the first or second motor.
        if(motor->channels == 1){
            motor->timer->Instance->CCR1 = ARR;
            motor->timer->Instance->CCR2 = ARR + doot;
        } else if(motor->channels == 2){
            motor->timer->Instance->CCR3 = ARR;
            motor->timer->Instance->CCR4 = ARR + doot;
        }else{
            // if neither return
            return;
        }
    // if duty cycle >=0
    } else{
        if(motor->channels == 1){
            motor->timer->Instance->CCR1 = ARR - doot;
            motor->timer->Instance->CCR2 = ARR;
        } else if(motor->channels == 2){
            motor->timer->Instance->CCR3 = ARR - doot;
            motor->timer->Instance->CCR4 = ARR;
        }else{
            return;
        }
    }
}

void motor_enable_disable(Motor* motor, uint8_t enable){
    // if user wants to enable motor
    if(enable == 1){
        motor->enable_flag = 1;
        // First retrieve ARR to set motor to brake mode
        uint32_t ARR = (uint32_t)(motor->timer->Init.Period + 1);

        // Now set the correct motor pair to brake mode.
        if(motor->channels == 1){
            motor->timer->Instance->CCR1 = ARR;
            motor->timer->Instance->CCR2 = ARR;
        } else if(motor->channels == 2){
            motor->timer->Instance->CCR3 = ARR;
            motor->timer->Instance->CCR4 = ARR;
        }else{
            return;
        }

        // set the motor's enable flag to 1
        motor->enable_flag = 1;

    // if user wants to disable motor
    } else if(enable == 0){
        motor->enable_flag = 0;
        if(motor->channels == 1){
            motor->timer->Instance->CCR1 = 0;
            motor->timer->Instance->CCR2 = 0;
        } else if(motor->channels == 2){
            motor->timer->Instance->CCR3 = 0;
            motor->timer->Instance->CCR4 = 0;
        }else{
            return;
        }

        // set the motor's enable flag to 0
        motor->enable_flag = 0;
    }
}
```

5.28 doxy_core/Src/pitch_encoder_handler.c File Reference

Implements the methods of the PitchEncoder class.

```
#include "pitch_encoder_handler.h"
```

Functions

- uint32_t get_pitch (PitchEncoder *p_enc)

5.28.1 Detailed Description

Implements the methods of the PitchEncoder class.

Date

May 30, 2024

Author

Jared Sinasohn

Definition in file pitch_encoder_handler.c.

5.28.2 Function Documentation

5.28.2.1 get_pitch()

```
uint32_t get_pitch (
            PitchEncoder * p_enc )
```

This function gets the current pitch based on the pitch selection knob

Parameters

p_enc	The pitch encoder object to read from

Returns

the current pitch, a number 0-11 mapped through the chromatic notes from A-Ab

read the current state of the encoder

store the delta of the encoder

add the delta to the pitch

if the pitch hasn't changed, just return the pitch

we can treat the pitch as a number between 0 and 11, which can underflow and overflow. We can run a similar algorithm to the delta() function in the encoder class to correct for this.

Definition at line 15 of file pitch_encoder_handler.c.

```
00015                                               {
00017     encoder_read_curr_state((p_enc->encoder));
00019     p_enc->delta = (int16_t)(p_enc->encoder->dx);
00021     p_enc->pitch += p_enc->delta;
00023     if(p_enc->delta == 0){
00024         return p_enc->pitch;
00025     }
00027     if(p_enc->pitch < 0){
00028         p_enc->pitch += 12;
00029     }
00030     if(p_enc->pitch >= 12){
00031         p_enc->pitch -= 12;
00032     }
00033     return p_enc->pitch;
00034 }
```

5.29 pitch_encoder_handler.c

Go to the documentation of this file.

```
00001
00008 #include "pitch_encoder_handler.h"
00009
00015 uint32_t get_pitch(PitchEncoder* p_enc){
00017     encoder_read_curr_state((p_enc->encoder));
00019     p_enc->delta = (int16_t)(p_enc->encoder->dx);
00021     p_enc->pitch += p_enc->delta;
00023     if(p_enc->delta == 0){
00024         return p_enc->pitch;
00025     }
00027     if(p_enc->pitch < 0){
00028         p_enc->pitch += 12;
00029     }
00030     if(p_enc->pitch >= 12){
00031         p_enc->pitch -= 12;
00032     }
00033     return p_enc->pitch;
00034 }
00035
```

Index