/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* File: main.cpp

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*---------------------------------------------------------------------------------------

\* INCLUDES

\*--------------------------------------------------------------------------------------\*/

#include "glo\_os\_sys.h"

#include "stm32f4xx.h"

#include "user\_leds.h"

#include "mpu6000.h"

#include "encoder.h"

#include "green\_leds.h"

#include "analog\_in.h"

#include "complementary\_filter.h"

#include "coordinate\_conversions.h"

#include "derivative\_filter.h"

#include "pid\_controller.h"

#include "user\_pb.h"

#include "usart.h"

#include "debug\_printf.h"

#include "digital\_out.h"

#include "digital\_in.h"

#include "physical\_constants.h"

#include <math.h>

// uncomment the define below to change orientation to the vertical (balance) orientation

#define VERTICAL\_CONFIGURATION

/\*---------------------------------------------------------------------------------------

\* LOCAL FUNCTION PROTOTYPES

\*--------------------------------------------------------------------------------------\*/

void drive\_task\_line\_arc(float);

/\*---------------------------------------------------------------------------------------

\* CONSTANTS

\*--------------------------------------------------------------------------------------\*/

#define WHEELRADIUS (0.03f) // meters

#define GEARRATIO (29.86f)

#define DISTPERCOUNT (12.0f) // meters per encoder tick

#define ENCODER2DIST (WHEELRADIUS \* 2.0f \* PI / GEARRATIO / DISTPERCOUNT) // distance [m] per count

#define ENCODER2THETA (6.283185f/29.86f/12.0f) // angle [rad] per count

/\*---------------------------------------------------------------------------------------

\* GLOBAL VARIABLES

\*--------------------------------------------------------------------------------------\*/

// Variables that come from flash\_params. Declared global because I am lazy

// these are really meant to be constants that are only changed at the

// the start by reading the flash params or by some calibration

float ACCEL\_SCALES[3] = {1.0f, 1.0f, 1.0f};

float ACCEL\_OFFSETS[3] = {0.0f, 0.0f, -0.8f};

float ENCODER\_SCALES[2] = {ENCODER2DIST, -ENCODER2DIST};

float MOTOR\_SIGNS[2]= {-1.0f, 1.0f};

float CURRENT\_SCALES[2] = {1.0f, 1.0f};

float CURRENT\_OFFSETS[2] = {0.0f, 0.0f};

float BATTERY\_SCALE = 4.286f;

float BATTERY\_OFFSET = 0.343f;

float WHEEL\_BASE = 0.089f;

bool mode=true;

float distance=0.0f;

/\*---------------------------------------------------------------------------------------

\* GLOBAL OBJECTS

\*--------------------------------------------------------------------------------------\*/

// global for easy tuning with debugger

static PidController yaw\_pid(2,20,.05,-.3,.3,-.8,.8 ); //2,20,.05,-.3,.3,-.8,.8 2.0, .75, 0.05, -0.3, 0.3, -0.5, 0.5

// Hardware instances used outside of main.

MPU6000 mpu; // used in filter task

AnalogIn analog; // used in high frequency control task

//FlashStorage flash\_storage;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Description: Sets everything up then enters an infinite while loop that is

\* the idle task.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main()

{

double start\_time; // seconds

// Set default state.

static glo\_modes\_t modes;

modes.op\_state = waiting\_for\_push\_button;

glo\_modes.publish(&modes);

configure\_glo\_os();

// Switch signs of motors depending on GPIO jumpers.

static DigitalIn left\_motor\_sign\_input(PA0,GPIO\_PuPd\_UP);

static DigitalIn right\_motor\_sign\_input(PC3,GPIO\_PuPd\_UP);

if ( left\_motor\_sign\_input.read() ) { MOTOR\_SIGNS[0] = 1.0f; }

else { MOTOR\_SIGNS[0] = -1.0f; }

if ( right\_motor\_sign\_input.read() ) { MOTOR\_SIGNS[1] = 1.0f; }

else { MOTOR\_SIGNS[1] = -1.0f; }

// Idle task

while (true)

{

static glo\_theta\_zero\_t theta\_zero;

static UserPB push\_button;

// Read in mode to be safe in case a task updated it.

// KLM: disabled to avoid constantly disabling interrupts in idle task.

// glo\_modes.read(&modes);

if ((push\_button.read()) && (modes.op\_state == waiting\_for\_push\_button))

{

debug\_printf("About to find zero tilt angle.");

modes.op\_state = getting\_theta\_zero;

glo\_modes.publish(&modes); // publish mode then wait at

start\_time = glo\_os->get\_time(); // least 0.1 seconds before

while ((glo\_os->get\_time() - start\_time) < 0.1); // zeroing the low pass filter

theta\_zero.theta = 0.0f; // to make sure the filter

glo\_theta\_zero.publish(&theta\_zero); // starts over at zero

}

else if (modes.op\_state == getting\_theta\_zero)

{

const double filter\_settle\_time = 3;

debug\_printf("Waiting %d sec for filter to settle.", (int32\_t)filter\_settle\_time);

start\_time = glo\_os->get\_time();

while ((glo\_os->get\_time() - start\_time) < filter\_settle\_time); // wait for the filter to settle

modes.op\_state = normal;

glo\_modes.publish(&modes);

debug\_printf("Finished waiting.");

}

else if (modes.op\_state == normal)

{

if ((push\_button.read()))

{

modes.op\_state = waiting\_for\_push\_button;

glo\_modes.publish(&modes);

start\_time = glo\_os->get\_time();

while ((glo\_os->get\_time() - start\_time) < 1); // wait for button release

}

}

} // end of idle task

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Description: A task function. Reads sensor data and publishes roll pitch yaw.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void complementary\_filter\_task\_callback(void)

{

const float deltaT = 0.002f; // 500Hz update rate

static ComplementaryFilter complementary\_filter;

static DigitalOut watch\_bit(PB5);

// define local instances of glo objects.

static glo\_raw\_accels\_t raw\_accels;

static glo\_gyros\_accels\_t gyros\_accels;

static glo\_quaternion\_t quaternion;

static glo\_roll\_pitch\_yaw\_t roll\_pitch\_yaw;

watch\_bit.set();

// read raw gyros and accels from MPU6000

float raw\_gyros[3];

mpu.readGyro(raw\_gyros);

mpu.readAccel(raw\_accels.accels);

// switch the axes (dependent on configuration) and apply accel calibration

#ifdef VERTICAL\_CONFIGURATION

gyros\_accels.accels[0] = -(raw\_accels.accels[2]\*ACCEL\_SCALES[2] + ACCEL\_OFFSETS[2]);

gyros\_accels.accels[1] = -(raw\_accels.accels[1]\*ACCEL\_SCALES[1] + ACCEL\_OFFSETS[1]);

gyros\_accels.accels[2] = -(raw\_accels.accels[0]\*ACCEL\_SCALES[0] + ACCEL\_OFFSETS[0]);

gyros\_accels.gyros[0] = -raw\_gyros[2];

gyros\_accels.gyros[1] = -raw\_gyros[1];

gyros\_accels.gyros[2] = -raw\_gyros[0];

#else

gyros\_accels.accels[0] = (raw\_accels.accels[0]\*ACCEL\_SCALES[0] + ACCEL\_OFFSETS[0]);

gyros\_accels.accels[1] = -(raw\_accels.accels[1]\*ACCEL\_SCALES[1] + ACCEL\_OFFSETS[1]);

gyros\_accels.accels[2] = -(raw\_accels.accels[2]\*ACCEL\_SCALES[2] + ACCEL\_OFFSETS[2]);

gyros\_accels.gyros[0] = raw\_gyros[0];

gyros\_accels.gyros[1] = -raw\_gyros[1];

gyros\_accels.gyros[2] = -raw\_gyros[2];

#endif

// run the complementary filter to calculate roll pitch yaw

complementary\_filter.update(deltaT, gyros\_accels.gyros, gyros\_accels.accels);

complementary\_filter.get\_state(quaternion.q);

quaternion\_2\_rpy(quaternion.q, roll\_pitch\_yaw.rpy);

// save the appropriate glo objects

glo\_raw\_accels.publish(&raw\_accels);

glo\_gyros\_accels.publish(&gyros\_accels);

glo\_quaternion.publish(&quaternion);

glo\_roll\_pitch\_yaw.publish(&roll\_pitch\_yaw);

watch\_bit.clear();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Description: A task function. Publishes torque command for each motor.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void main\_control\_task\_callback(void)

{

static Encoder left\_encoder(EncoderA);

static Encoder right\_encoder(EncoderB);

const float deltaT = 0.001f;

static DerivativeFilter left\_deriv(deltaT, 20.0f , 0.707f);

static DerivativeFilter right\_deriv(deltaT, 20.0f , 0.707f);

static DerivativeFilter pos\_cmd\_deriv(deltaT, 50.0f , 0.707f);

static DerivativeFilter theta\_cmd\_deriv(deltaT, 100.0f , 0.707f);

static DerivativeFilter beta\_deriv(deltaT, 10.0f , 0.707f);

static DigitalOut watch\_bit(PB4);

static float distance\_command = 0.0f;

static float yaw\_command = 0.0f;

// define local instances of glo objects

static glo\_modes\_t modes;

static glo\_gyros\_accels\_t gyros\_accels;

static glo\_roll\_pitch\_yaw\_t roll\_pitch\_yaw;

static glo\_theta\_zero\_t theta\_zero;

static glo\_odometry\_t odometry;

static glo\_motion\_commands\_t motion\_commands;

watch\_bit.set();

// // read in glo object data

// glo\_modes.read(&modes);

// glo\_gyros\_accels.read(&gyros\_accels);

// glo\_roll\_pitch\_yaw.read(&roll\_pitch\_yaw);

// glo\_theta\_zero.read(&theta\_zero);

// glo\_motion\_commands.read(&motion\_commands);

// distance\_command += motion\_commands.speed \* deltaT;

// yaw\_command += motion\_commands.omega \* deltaT;

// Convert encoder measurements to odometry data.

odometry.left\_distance = ENCODER\_SCALES[0] \* left\_encoder.read();

odometry.right\_distance = ENCODER\_SCALES[1] \* right\_encoder.read();

odometry.avg\_distance = (odometry.left\_distance + odometry.right\_distance) / 2.0f;

odometry.yaw = (odometry.left\_distance - odometry.right\_distance) / WHEEL\_BASE;

odometry.left\_speed = left\_deriv.calculate(odometry.left\_distance);

odometry.right\_speed = right\_deriv.calculate(odometry.right\_distance);

odometry.avg\_speed = (odometry.left\_speed + odometry.right\_speed) / 2.0f;

// drive\_task\_line\_arc(odometry.avg\_distance);

glo\_modes.read(&modes);

glo\_gyros\_accels.read(&gyros\_accels);

glo\_roll\_pitch\_yaw.read(&roll\_pitch\_yaw);

glo\_theta\_zero.read(&theta\_zero);

glo\_motion\_commands.read(&motion\_commands);

distance\_command += motion\_commands.speed \* deltaT;

yaw\_command += motion\_commands.omega \* deltaT;

// Set desired tilt to the tilt when initializing.

float theta\_cmd = theta\_zero.theta;

// Measure states. theta = tilt and beta = wheel angle

// Need to add tilt to encoder measurement to get absolute wheel angle

float theta = roll\_pitch\_yaw.rpy[1];

float thetad = gyros\_accels.gyros[1];

float beta\_relative = odometry.avg\_distance / WHEELRADIUS;

float beta = beta\_relative + theta;

float betad = beta\_deriv.calculate(beta\_relative) + thetad;

// Multiply states by feedback gains to calculate current command.

static const float K[4] = { -1.656553f, -0.1296066f, 0.03936147f, 0.02009268f };

float beta\_command = distance\_command / WHEELRADIUS;

float feedback\_current = theta\*K[0] + thetad\*K[1] + (beta-beta\_command)\*K[2] + betad\*K[3];

float theta\_error = theta\_cmd - roll\_pitch\_yaw.rpy[1] + theta\_zero.theta;

// Run yaw controller to calculate desired difference in current between motors.

float yaw\_error = (yaw\_command - odometry.yaw); //odometry.yaw); roll\_pitch\_yaw.rpy[2]);

float delta\_current = yaw\_pid.calculate(yaw\_error, -gyros\_accels.gyros[2], deltaT);

float current\_command = feedback\_current;

bool not\_in\_control\_mode = (modes.op\_state != normal);

bool fallen\_down = (modes.op\_state == normal) && ((theta\_error > 0.8f) || (theta\_error < -0.8f));

if (fallen\_down || not\_in\_control\_mode)

{

current\_command = 0.0f;

yaw\_pid.resetIntegral();

// Keep commands in sync with state to keep from building up large error.

distance\_command = odometry.avg\_distance;

yaw\_command = odometry.yaw;

motion\_commands.speed = 0.0f;

motion\_commands.omega = 0.0f;

}

if (modes.op\_state == getting\_theta\_zero)

{

// Finding initialization tilt using a first order low pass filter with

// time constant of 2.5 seconds

const float alpha = deltaT / 2.5f;

theta\_zero.theta += alpha \* (roll\_pitch\_yaw.rpy[1] - theta\_zero.theta);

glo\_theta\_zero.publish(&theta\_zero);

}

glo\_current\_commands\_t current\_commands;

current\_commands.amps[0] = (current\_command + delta\_current) \* MOTOR\_SIGNS[0];

current\_commands.amps[1] = (current\_command - delta\_current) \* MOTOR\_SIGNS[1];

glo\_current\_commands.publish(&current\_commands);

// save the appropriate glo objects

glo\_odometry.publish(&odometry);

glo\_motion\_commands.publish(&motion\_commands);

watch\_bit.clear();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Description: Set LEDs depending on battery state and possibly other things.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void leds\_task\_callback(void)

{

static UserLeds leds;

static GreenLeds gleds;

static DigitalOut watch\_bit(PB9);

watch\_bit.set();

// Read glo object data into local instances.

glo\_modes\_t modes;

glo\_modes.read(&modes);

glo\_raw\_analog\_t analog;

glo\_raw\_analog.read(&analog);

float batt\_voltage = BATTERY\_SCALE \* analog.voltages[8] + BATTERY\_OFFSET;

if ((!modes.params\_set)

&& (modes.op\_state == normal))

{

leds.toggle(orange);

leds.toggle(red);

}

// use green & yellow for batt voltage. Solid green = good, solid yellow = bad!

if (batt\_voltage > 8.1f) {leds.set(green); leds.clear(yellow); }

else if (batt\_voltage > 7.4f) {leds.toggle(green); leds.clear(yellow); }

else if (batt\_voltage > 6.7f) {leds.toggle(green); leds.toggle(yellow);}

else if (batt\_voltage > 6.0f) {leds.toggle(green); leds.set(yellow); }

else {leds.clear(green); leds.set(yellow); }

// Binary counter on green LEDs

static uint8\_t count = 0;

count++;

gleds.set(count);

watch\_bit.clear();

}

void drive\_task\_line\_arc(float avg\_distance)

{

static glo\_motion\_commands\_t motion\_commands;

static float radius=.25f;

static float curve=(315.0f/180.0f)\*PI\*radius;

static float straight=.25f;

static float sign\_direction=1.0f;

static float number=0.0f;

if (number>=1.0f)

{

curve=(317.0f/180.0f)\*PI\*radius;

straight=.245f;

}

distance=avg\_distance-(straight+curve)\*number;

//mode = true -> straight mode = false -> arc

if(mode==true)

{

motion\_commands.speed=.15f;

motion\_commands.omega=0.0f;

if(distance>=straight)

{

mode=false;

}

}

else if (mode==false)

{

motion\_commands.speed=.15f;

motion\_commands.omega=sign\_direction\*(.15f/radius); //sign\_direction\*(.15f/radius);

if(distance>=curve)

{

mode=true;

sign\_direction\*=-1.0f;

number+=1.0f;

}

}

glo\_motion\_commands.publish(&motion\_commands);

}