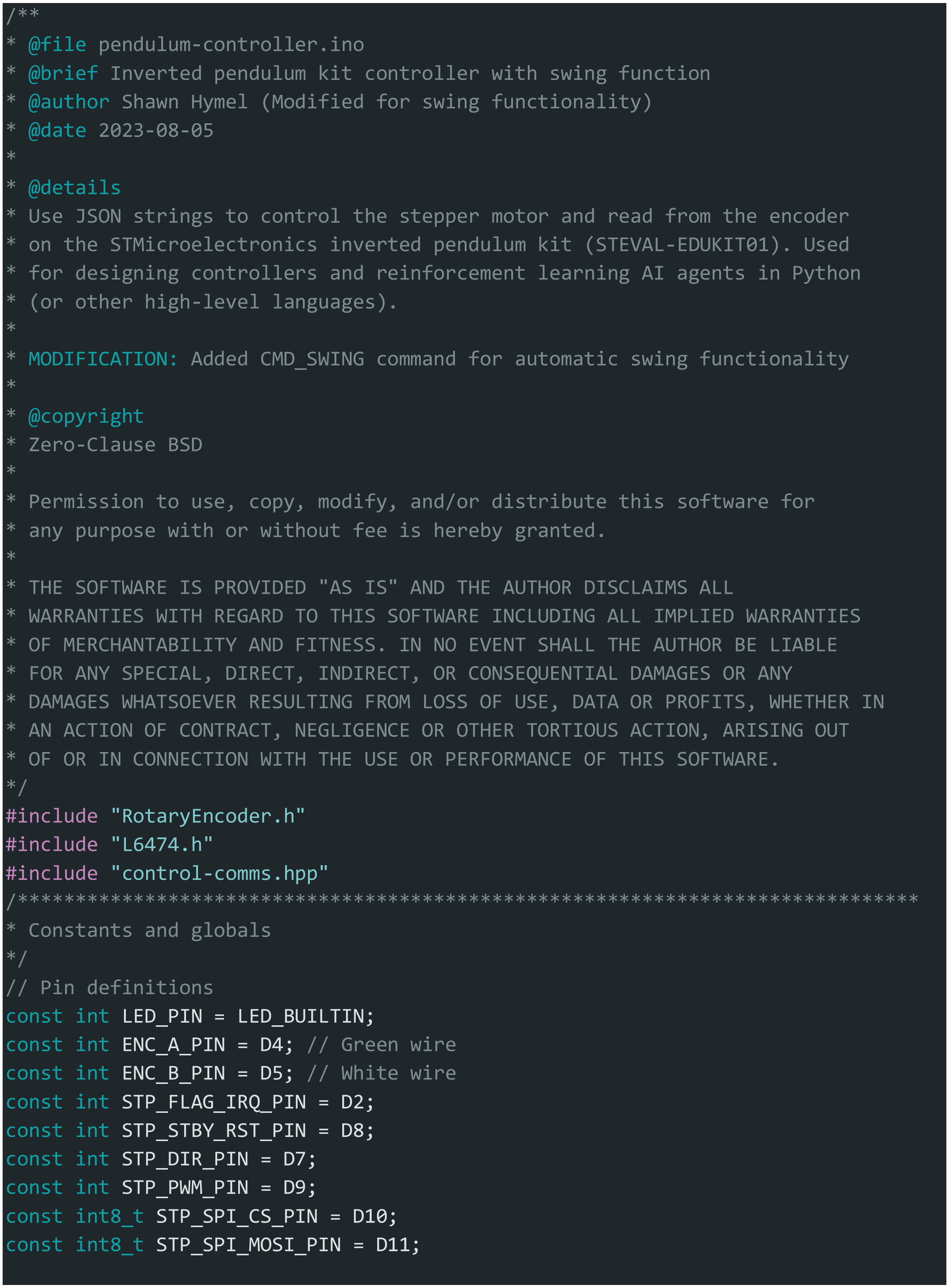
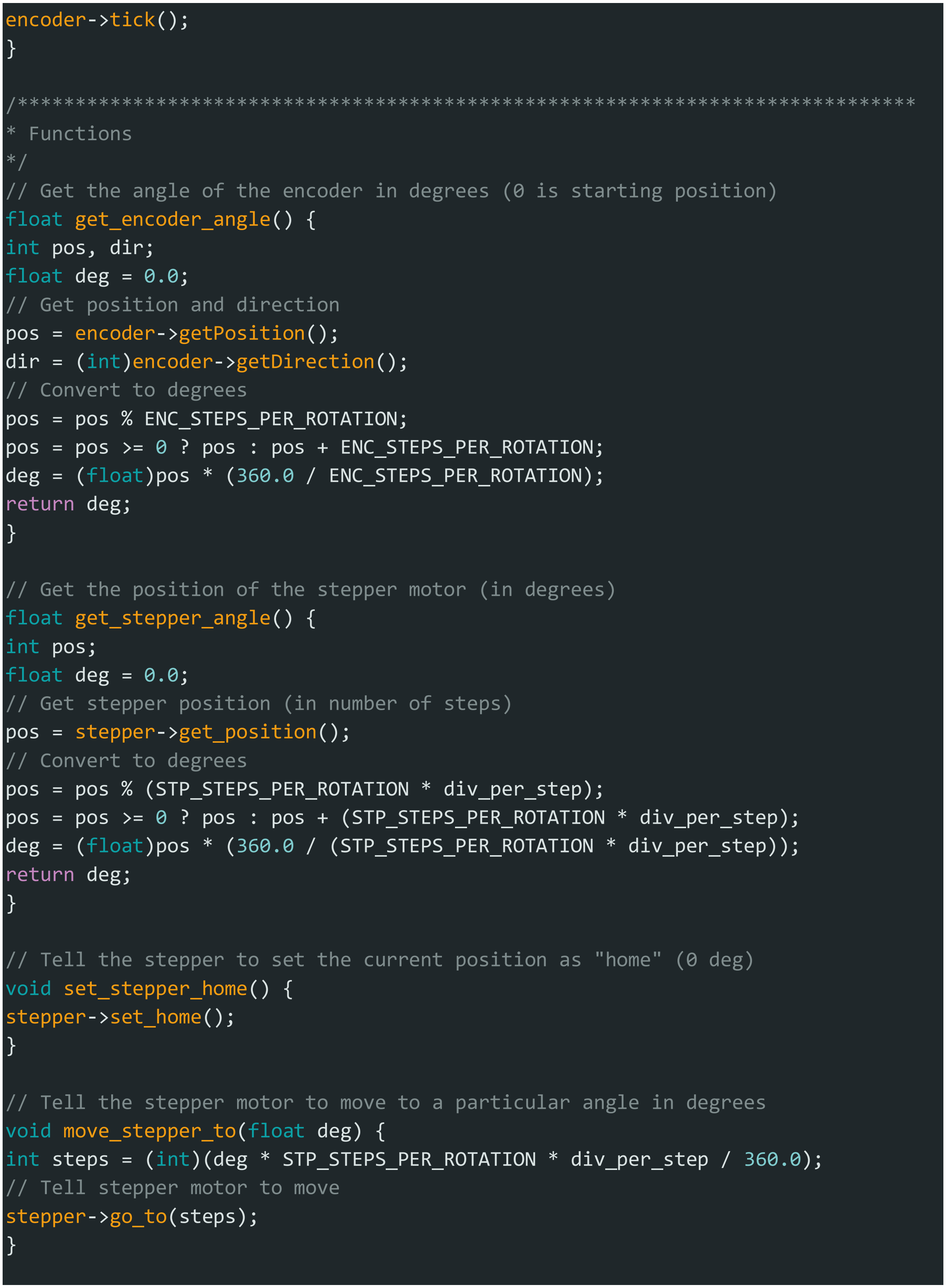


```
/**
* @file pendulum-controller.ino
* @brief Inverted pendulum kit controller with swing function
* @author Shawn Hymel (Modified for swing functionality)
* @date 2023-08-05
*
* @details
* Use JSON strings to control the stepper motor and read from the encoder
* on the STMicroelectronics inverted pendulum kit (STEVAL-EDUKIT01). Used
* for designing controllers and reinforcement learning AI agents in Python
* (or other high-level languages).
*
* MODIFICATION: Added CMD_SWING command for automatic swing functionality
*
* @copyright
* Zero-Clause BSD
*
* Permission to use, copy, modify, and/or distribute this software for
* any purpose with or without fee is hereby granted.
*
* THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
* WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE
* FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY
* DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
* AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
* OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/
#include "RotaryEncoder.h"
#include "L6474.h"
#include "control-comms.hpp"
/******************************************************************************
* Constants and globals
*/
// Pin definitions
const int LED_PIN = LED_BUILTIN;
const int ENC_A_PIN = D4; // Green wire
const int ENC_B_PIN = D5; // White wire
const int STP_FLAG_IRQ_PIN = D2;
const int STP_STBY_RST_PIN = D8;
const int STP_DIR_PIN = D7;
const int STP_PWM_PIN = D9;
const int8_t STP_SPI_CS_PIN = D10;
const int8_t STP_SPI_MOSI_PIN = D11;
const int8_t STP_SPI_MISO_PIN = D12;
const int8_t STP_SPI_SCK_PIN = D13;
// Communication constants
static const unsigned int BAUD_RATE = 500000;
static const ControlComms::DebugLevel CTRL_DEBUG = ControlComms::DEBUG_ERROR;
static constexpr size_t NUM_ACTIONS = 1;
static constexpr size_t NUM_OBS = 2;
static const unsigned int STATUS_OK = 0;
static const unsigned int STATUS_STP_MOVING = 1;
static const unsigned int CMD_SET_HOME = 0;
static const unsigned int CMD_MOVE_TO = 1;
static const unsigned int CMD_MOVE_BY = 2;
static const unsigned int CMD_SET_STEP_MODE = 3;
// MODIFICACIÓN: Nuevo comando para swing automático
static const unsigned int CMD_SWING = 4; // *** LÍNEA MODIFICADA: Comando swing ***
// Stepper and encoder constants
const int ENC_STEPS_PER_ROTATION = 1200;
const int STP_STEPS_PER_ROTATION = 200; // 200 full steps, must multiply by div_per_step!

// MODIFICACIÓN: Constantes para el swing
const float SWING_ANGLE = 180.0; // *** LÍNEA MODIFICADA: Ángulo de swing (puedes modificar este valor) ***
const unsigned long SWING_DELAY = 1500; // *** LÍNEA MODIFICADA: Delay entre movimientos (1 segundo) ***
const float TARGET_ANGLE = 180.0; // *** LÍNEA MODIFICADA: Ángulo objetivo del encoder ***

// Stepper config
L6474_init_t stepper_config = {
15000, // Acceleration rate in pps^2. Range: (0..+inf)
10000, // Deceleration rate in pps^2. Range: (0..+inf)
3000, // Maximum speed in pps. Range: (30..10000]
1000, // Minimum speed in pps. Range: [30..10000)
300, // Torque regulation current in mA. Range: 31.25mA to 4000mA
L6474_OCD_TH_750mA, // Overcurrent threshold (OCD_TH register)
L6474_CONFIG_OC_SD_ENABLE, // Overcurrent shutwdown (OC_SD field of CONFIG register)
L6474_CONFIG_EN_TQREG_TVAL_USED, // Torque regulation method (EN_TQREG field of CONFIG register)
L6474_STEP_SEL_1_16, // Step selection (STEP_SEL field of STEP_MODE register)
L6474_SYNC_SEL_1_2, // Sync selection (SYNC_SEL field of STEP_MODE register)
L6474_FAST_STEP_12us, // Fall time value (T_FAST field of T_FAST register). Range: 2us to 32us
```

```
L6474_TOFF_FAST_8us, // Maximum fast decay time (T_OFF field of T_FAST register).
Range: 2us to 32us
3, // Minimum ON time in us (TON_MIN register). Range: 0.5us to 64us
21, // Minimum OFF time in us (TOFF_MIN register). Range: 0.5us to 64us
L6474_CONFIG_TOFF_044us, // Target Switching Period (field TOFF of CONFIG
register)
L6474_CONFIG_SR_320V_us, // Slew rate (POW_SR field of CONFIG register)
L6474_CONFIG_INT_16MHZ, // Clock setting (OSC_CLK_SEL field of CONFIG register)
L6474_ALARM_EN_OVERCURRENT |
L6474_ALARM_EN_THERMAL_SHUTDOWN |
L6474_ALARM_EN_THERMAL_WARNING |
L6474_ALARM_EN_UNDERVOLTAGE |
L6474_ALARM_EN_SW_TURN_ON |
L6474_ALARM_EN_WRONG_NPERF_CMD // Alarm (ALARM_EN register)
};

// Globals
RotaryEncoder *encoder = nullptr;
volatile int led_state = 0;
SPIClass dev_spi(STP_SPI_MOSI_PIN, STP_SPI_MISO_PIN, STP_SPI_SCK_PIN);
L6474 *stepper;
ControlComms ctrl;
unsigned int div_per_step = 16;

/******************************************************************************
* Interrupt service routines (ISRs)
*/
// Stepper interrupt service routine (timer)
void stepperISR(void) {
// Set ISR flag in stepper controller
stepper->isr_flag = TRUE;
// Read the status register
unsigned int status = stepper->get_status();
// If NOTPERF_CMD flag is set, the SPI command cannot be performed
if ((status & L6474_STATUS_NOTPERF_CMD) == L6474_STATUS_NOTPERF_CMD) {
Serial.println(" WARNING: FLAG interrupt triggered. Non-performable " \
"command detected when updating L6474's registers while " \
"not in HiZ state.");
}
// Reset ISR flag in stepper controller
stepper->isr_flag = FALSE;
}

// Encoder interrupt service routine (pin change): check state
void encoderISR() {
```



```
encoder->tick();
}

/*******************************************************************************
* Functions
*/
// Get the angle of the encoder in degrees (0 is starting position)
float get_encoder_angle() {
int pos, dir;
float deg = 0.0;
// Get position and direction
pos = encoder->getPosition();
dir = (int)encoder->getDirection();
// Convert to degrees
pos = pos % ENC_STEPS_PER_ROTATION;
pos = pos >= 0 ? pos : pos + ENC_STEPS_PER_ROTATION;
deg = (float)pos * (360.0 / ENC_STEPS_PER_ROTATION);
return deg;
}

// Get the position of the stepper motor (in degrees)
float get_stepper_angle() {
int pos;
float deg = 0.0;
// Get stepper position (in number of steps)
pos = stepper->get_position();
// Convert to degrees
pos = pos % (STP_STEPS_PER_ROTATION * div_per_step);
pos = pos >= 0 ? pos : pos + (STP_STEPS_PER_ROTATION * div_per_step);
deg = (float)pos * (360.0 / (STP_STEPS_PER_ROTATION * div_per_step));
return deg;
}

// Tell the stepper to set the current position as "home" (0 deg)
void set_stepper_home() {
stepper->set_home();
}

// Tell the stepper motor to move to a particular angle in degrees
void move_stepper_to(float deg) {
int steps = (int)(deg * STP_STEPS_PER_ROTATION * div_per_step / 360.0);
// Tell stepper motor to move
stepper->go_to(steps);
}
```

```
// Tell the stepper motor to move by a particular angle in degrees
void move_stepper_by(float deg) {
StepperMotor::direction_t stp_dir = StepperMotor::FWD;
int steps = (int)(deg * STP_STEPS_PER_ROTATION * div_per_step / 360.0);
// Use direction and absolute step counts
if (steps < 0) {
steps = -1 * steps;
stp_dir = StepperMotor::BWD;
}
// Tell stepper motor to move
stepper->move(stp_dir, steps);
}

// Set step mode according to p. 38 in the L6474 datasheet (modes 0..4)
void set_step_mode(int mode) {
switch (mode) {
case 0:
stepper->set_step_mode(StepperMotor::STEP_MODE_FULL);
div_per_step = 1;
break;
case 1:
stepper->set_step_mode(StepperMotor::STEP_MODE_HALF);
div_per_step = 2;
break;
case 2:
stepper->set_step_mode(StepperMotor::STEP_MODE_1_4);
div_per_step = 4;
break;
case 3:
stepper->set_step_mode(StepperMotor::STEP_MODE_1_8);
div_per_step = 8;
break;
case 4:
stepper->set_step_mode(StepperMotor::STEP_MODE_1_16);
div_per_step = 16;
break;
default:
break;
}
}

// MODIFICACIÓN: Función para realizar swing automático
// *** LÍNEAS MODIFICADAS: Nueva función swing_to_target ***
void swing_to_target() {
float encoder_angle = get_encoder_angle();
```

```
// Normalizar el ángulo del encoder para manejar el cruce de 0°/360°
float normalized_angle = encoder_angle;
if (normalized_angle > 270) {
normalized_angle = normalized_angle - 360; // Convertir 359° a -1°
}
// Calcular diferencia con el objetivo (180°)
float angle_diff = abs(TARGET_ANGLE - normalized_angle);
if (normalized_angle > 180) {
angle_diff = abs(TARGET_ANGLE - (normalized_angle - 360));
}
// Repetir swing hasta alcanzar el ángulo objetivo
while (angle_diff > 5.0) { // Tolerancia de 5 grados
// Movimiento a la derecha
move_stepper_by(SWING_ANGLE);
// Esperar a que termine el movimiento
while (stepper->get_device_state() != INACTIVE) {
delay(10);
}
// Esperar tiempo configurado
delay(SWING_DELAY);
// Movimiento a la izquierda
move_stepper_by(-SWING_ANGLE);
// Esperar a que termine el movimiento
while (stepper->get_device_state() != INACTIVE) {
delay(10);
}
// Esperar tiempo configurado
delay(SWING_DELAY);
// Actualizar lectura del encoder y recalcular diferencia
encoder_angle = get_encoder_angle();
normalized_angle = encoder_angle;
if (normalized_angle > 270) {
normalized_angle = normalized_angle - 360;
}
angle_diff = abs(TARGET_ANGLE - normalized_angle);
if (normalized_angle > 180) {
angle_diff = abs(TARGET_ANGLE - (normalized_angle - 360));
}
}
}

/***************************************************************************
* Main
*/
void setup() {
```

```
// Configure pins
pinMode(LED_PIN, OUTPUT);
pinMode(D4, INPUT_PULLUP);
pinMode(D5, INPUT_PULLUP);
// Initialize our communication interface
Serial.begin(BAUD_RATE);
ctrl.init(Serial, CTRL_DEBUG);
// Configure encoder
encoder = new RotaryEncoder(
ENC_A_PIN,
ENC_B_PIN,
RotaryEncoder::LatchMode::TWO03
);
// Configure encoder interrupts
attachInterrupt(digitalPinToInterrupt(ENC_A_PIN), encoderISR, CHANGE);
attachInterrupt(digitalPinToInterrupt(ENC_B_PIN), encoderISR, CHANGE);
// Initialize stepper motor control
stepper = new L6474(
STP_FLAG_IRQ_PIN,
STP_STBY_RST_PIN,
STP_DIR_PIN,
STP_PWM_PIN,
STP_SPI_CS_PIN,
&dev_spi
);
if (stepper->init(&stepper_config) != COMPONENT_OK) {
Serial.println("ERROR: Could not initialize stepper driver");
while(1);
}
// Attach and enable stepper motor interrupt handlers
stepper->attach_flag_irq(&stepperISR);
stepper->enable_flag_irq();
// Set current position as home
stepper->set_home();
}

void loop() {
int command;
float action[NUM_ACTIONS];
int status;
float observation[NUM_OBS];
ControlComms::StatusCode rx_code;
// Receive
rx_code = ctrl.receive_action<NUM_ACTIONS>(&command, action);
if (rx_code == ControlComms::OK) {
```

```
// Move the stepper as requested
switch (command) {
case CMD_SET_HOME:
set_stepper_home();
break;
case CMD_MOVE_TO:
move_stepper_to(action[0]);
break;
case CMD_MOVE_BY:
move_stepper_by(action[0]);
break;
case CMD_SET_STEP_MODE:
set_step_mode((unsigned int)action[0]);
set_stepper_home();
break;
// MODIFICACIÓN: Nuevo caso para comando swing
case CMD_SWING: // *** LÍNEA MODIFICADA: Nuevo case para swing ***
swing_to_target();
break;
default:
break;
}
// Read encoder and stepper angles (in degrees)
observation[0] = get_encoder_angle();
observation[1] = get_stepper_angle();
// Determine motor status
if (stepper->get_device_state() != INACTIVE) {
status = STATUS_STP_MOVING;
} else {
status = STATUS_OK;
}
// Send back observation
ctrl.send_observation(status, millis(), false, observation, NUM_OBS);
// Handle receiver error (ignore "RX_EMPTY" case)
} else if (rx_code == ControlComms::ERROR) {
Serial.println("Error receiving actions");
}
}
```