```
/**
* @file pendulum-controller-stepper-swingup.ino
* @brief Controlador de péndulo invertido - Swing-up con motor paso a paso
* @author Versión corregida para swing-up con motor stepper únicamente
* @date 2025-08-20
*
* @details
* CORRECCIONES PRINCIPALES:
* 1. Swing-up realizado únicamente con motor paso a paso (stepper)
* 2. Encoder solo como sensor para medir ángulo del péndulo
* 3. Movimientos rápidos de swing-up: impulso fuerte -> parada -> espera ->
impulso contrario
* 4. Transición automática al control cuando péndulo alcanza ~180°
* 5. Control PID/LQR sobre la función de transferencia del sistema
*/
#include "RotaryEncoder.h"
#include "L6474.h"
#include "control-comms.hpp"
#include <math.h>

/******************************************************************************
* Constants and globals
*/
// Pin definitions
const int LED_PIN = LED_BUILTIN;
const int ENC_A_PIN = D4; // Green wire - Encoder (péndulo sensor)
const int ENC_B_PIN = D5; // White wire - Encoder (péndulo sensor)

// Stepper motor pins
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 = 6;
static constexpr size_t NUM_OBS = 6;

// Command definitions
```

```
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;
static const unsigned int CMD_SELECT_CONTROLLER = 4;
static const unsigned int CMD_SET_PID_GAINS = 5;
static const unsigned int CMD_SET_LQR_GAINS = 6;
static const unsigned int CMD_START_CONTROL = 7;
static const unsigned int CMD_STOP_CONTROL = 8;

// Status codes
static const unsigned int STATUS_OK = 0;
static const unsigned int STATUS_STP_MOVING = 1;
static const unsigned int STATUS_SWING_UP = 2;
static const unsigned int STATUS_CONTROL_ACTIVE = 3;
static const unsigned int STATUS_UPRIGHT_ACHIEVED = 4;

// Physical constants
const float SAMPLE_TIME = 0.004; // 4ms sampling time
const float l = 0.258; // Length parameter
const float r = 0.141; // Rotor parameter
const float g = 9.806; // Gravity
const float sigma = g/(l*pow(2*PI*1.19, 2));

// Encoder and stepper constants
const int ENC_STEPS_PER_ROTATION = 1200;
const int STP_STEPS_PER_ROTATION = 200;

// Control parameters
const float UPRIGHT_THRESHOLD = 15.0; // degrees from 180
const float SETPOINT = 180.0; // Upright position in degrees

// PARÁMETROS DE SWING-UP CON STEPPER OPTIMIZADOS
const float SWING_IMPULSE_ANGLE = 30.0; // degrees - ángulo de impulso por
movimiento
const float SWING_IMPULSE_SPEED = 500.0; // steps/sec - velocidad del impulso
const float SWING_WAIT_TIME = 0.8; // seconds - tiempo de espera entre impulsos
const float SWING_MAX_DURATION = 30.0; // seconds - tiempo máximo
const int SWING_MAX_CYCLES = 50; // máximo número de ciclos
const float SWING_ENERGY_THRESHOLD = 10.0; // umbral de energía para incrementar
impulso

// Controller selection
typedef enum {
    CONTROLLER_PID = 0,
```

```
    CONTROLLER_LQR = 1
} ControllerType;

// System states
typedef enum {
    STATE_IDLE = 0,
    STATE_SWING_UP = 1,
    STATE_CONTROL = 2,
    STATE_STOPPED = 3
} SystemState;

// Swing-up sub-states
typedef enum {
    SWING_IMPULSE = 0,
    SWING_WAIT = 1,
    SWING_DIRECTION_CHANGE = 2
} SwingState;

// Global variables
RotaryEncoder *encoder = nullptr;
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;

// Control variables
ControllerType current_controller = CONTROLLER_PID;
SystemState system_state = STATE_IDLE;

// PID gains
float K_p = 0.1, K_i = 0.0008, K_d = 0.012;
float pid_error_prev = 0.0, pid_integral = 0.0;

// LQR gains
float K_lqr[5] = {0.0, 0.0, 0.0, 0.0, 0.0};

// State variables
float phi = 0.0; // Rotor angle (degrees)
float phi_prev = 0.0;
float dphi = 0.0; // Rotor angular velocity
float theta = 0.0; // Pendulum angle (degrees)
float theta_prev = 0.0;
float dtheta = 0.0; // Pendulum angular velocity
float integral_error = 0.0;
```

```
// Swing-up variables
float phi_reference = 0.0; // Referencia del stepper cuando se alcanza 180°
unsigned long swing_start_time = 0;
unsigned long swing_phase_start_time = 0;
int swing_cycle_count = 0;
bool upright_achieved = false;
SwingState swing_state = SWING_IMPULSE;
bool swing_direction = true; // true = positive direction, false = negative
float current_impulse_angle = SWING_IMPULSE_ANGLE;
float theta_max_achieved = 0.0; // Máximo ángulo alcanzado para ajuste automático

// Energy estimation variables
float pendulum_energy = 0.0;
float energy_target = 50.0; // Energía objetivo para swing-up

// Timing variables
unsigned long current_time = 0;
unsigned long prev_time = 0;
unsigned long control_loop_timer = 0;
const unsigned long CONTROL_INTERVAL = 4; // 4ms = 250Hz

// Debug variables
unsigned long last_debug_print = 0;
const unsigned long DEBUG_INTERVAL = 500; // Print debug cada 500ms

// Stepper configuration
L6474_init_t stepper_config = {
    10000, 10000, 5000, 1000, 300,
    L6474_OCD_TH_750mA, L6474_CONFIG_OC_SD_ENABLE,
    L6474_CONFIG_EN_TQREG_TVAL_USED, L6474_STEP_SEL_1_16,
    L6474_SYNC_SEL_1_2, L6474_FAST_STEP_12us, L6474_TOFF_FAST_8us,
    3, 21, L6474_CONFIG_TOFF_044us, L6474_CONFIG_SR_320V_us,
    L6474_CONFIG_INT_16MHZ,
    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
};

/*******************************************************************************
* Interrupt service routines
*/
void stepperISR(void) {
    stepper->isr_flag = TRUE;
    unsigned int status = stepper->get_status();
    if ((status & L6474_STATUS_NOTPERF_CMD) == L6474_STATUS_NOTPERF_CMD) {
```

```
        // Serial.println("WARNING: FLAG interrupt triggered.");
    }
    stepper->isr_flag = FALSE;
}

void encoderISR() {
    encoder->tick();
}

/*****************************************************************************
* Utility functions
*/
float get_encoder_angle() {
    int pos = encoder->getPosition();
    pos = pos % ENC_STEPS_PER_ROTATION;
    pos = pos >= 0 ? pos : pos + ENC_STEPS_PER_ROTATION;
    return (float)pos * (360.0 / ENC_STEPS_PER_ROTATION);
}

float get_stepper_angle() {
    int pos = stepper->get_position();
    pos = pos % (STP_STEPS_PER_ROTATION * div_per_step);
    pos = pos >= 0 ? pos : pos + (STP_STEPS_PER_ROTATION * div_per_step);
    return (float)pos * (360.0 / (STP_STEPS_PER_ROTATION * div_per_step));
}

void set_stepper_home() {
    stepper->set_home();
    phi_reference = 0.0;
    upright_achieved = false;
    Serial.println("Stepper home position set");
}

void move_stepper_to(float deg) {
    int steps = (int)(deg * STP_STEPS_PER_ROTATION * div_per_step / 360.0);
    stepper->go_to(steps);
}

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);
    if (steps < 0) {
        steps = -1 * steps;
        stp_dir = StepperMotor::BWD;
    }
```

```
    stepper->move(stp_dir, steps);
}

void move_stepper_fast(float deg, float speed) {
    // Configurar velocidad alta para swing-up
    stepper->set_max_speed(speed);
    stepper->set_acceleration(2000); // Aceleración alta para movimientos rápidos
    move_stepper_by(deg);
}

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;
    }
}

/******************************************************************************
* State estimation
*/
void update_states() {
    current_time = millis();
    float dt = (current_time - prev_time) / 1000.0;
    if (dt <= 0.0) dt = SAMPLE_TIME;

    // Update angles
    phi_prev = phi;
    theta_prev = theta;
    phi = get_stepper_angle();
    theta = get_encoder_angle();

    // Calculate derivatives
    dphi = (phi - phi_prev) / dt;
    dtheta = (theta - theta_prev) / dt;

    // Limit velocities to avoid noise spikes
```

```
    if (abs(dphi) > 1000.0) dphi = 0.0;
    if (abs(dtheta) > 1000.0) dtheta = 0.0;

    // Calculate pendulum energy for swing-up control
    // E = 1/2 * I * w^2 + m*g*l*(1-cos(theta))
    float theta_rad = theta * PI / 180.0;
    float dtheta_rad = dtheta * PI / 180.0;
    pendulum_energy = 0.5 * dtheta_rad * dtheta_rad + sigma * (1.0 -
cos(theta_rad));

    // Update integral error for control
    float error = SETPOINT - theta;
    integral_error += error * dt;

    // Limit integral windup
    if (integral_error > 100.0) integral_error = 100.0;
    if (integral_error < -100.0) integral_error = -100.0;

    // Track maximum angle achieved during swing-up
    if (system_state == STATE_SWING_UP) {
        float angle_from_bottom = theta;
        if (angle_from_bottom > 180.0) angle_from_bottom = 360.0 -
angle_from_bottom;
        if (angle_from_bottom > theta_max_achieved) {
            theta_max_achieved = angle_from_bottom;
        }
    }

    prev_time = current_time;
}

/*******************************************************************************
* Control algorithms
*/
float compute_pid_control() {
    float error = SETPOINT - theta;
    pid_integral += error * SAMPLE_TIME;

    // Limit integral windup
    if (pid_integral > 100.0) pid_integral = 100.0;
    if (pid_integral < -100.0) pid_integral = -100.0;

    float derivative = (error - pid_error_prev) / SAMPLE_TIME;
    float output = K_p * error + K_i * pid_integral + K_d * derivative;
    pid_error_prev = error;
```

```
    // Limit output
    if (output > 45.0) output = 45.0;
    if (output < -45.0) output = -45.0;

    return output;
}

float compute_lqr_control() {
    // State vector: [phi_error, dphi, theta_error, dtheta, integral_error]
    float phi_error_rad = (phi - phi_reference) * PI / 180.0;
    float dphi_rad = dphi * PI / 180.0;
    float theta_error_rad = (theta - SETPOINT) * PI / 180.0;
    float dtheta_rad = dtheta * PI / 180.0;

    // LQR control law: u = -K * x
    float control = -(K_lqr[0] * phi_error_rad +
                     K_lqr[1] * dphi_rad +
                     K_lqr[2] * theta_error_rad +
                     K_lqr[3] * dtheta_rad +
                     K_lqr[4] * integral_error);

    // Convert back to degrees and limit
    control = control * 180.0 / PI;
    if (control > 45.0) control = 45.0;
    if (control < -45.0) control = -45.0;

    return control;
}

bool is_pendulum_upright() {
    float distance_from_180 = abs(theta - 180.0);
    if (distance_from_180 > 180.0) {
        distance_from_180 = 360.0 - distance_from_180;
    }
    return distance_from_180 <= UPRIGHT_THRESHOLD;
}

/******************************************************************************
* SWING-UP ALGORITHM CON MOTOR STEPPER
*/
void reset_swing_up_variables() {
    swing_cycle_count = 0;
    swing_start_time = millis();
    swing_phase_start_time = millis();
```

```
    upright_achieved = false;
    swing_state = SWING_IMPULSE;
    swing_direction = true;
    current_impulse_angle = SWING_IMPULSE_ANGLE;
    theta_max_achieved = 0.0;

    // Configurar stepper para swing-up (velocidades altas)
    stepper->set_max_speed(SWING_IMPULSE_SPEED);
    stepper->set_acceleration(2000);

    Serial.println("=== INICIANDO SWING-UP CON MOTOR STEPPER ===");
    Serial.print("Ángulo inicial péndulo: "); Serial.print(theta, 1);
Serial.println("°");
    Serial.print("Posición inicial stepper: "); Serial.print(phi, 1);
Serial.println("°");
    Serial.println("ESTRATEGIA: Impulsos rápidos del stepper con paradas y
esperas");
    Serial.print("Ángulo de impulso: ±"); Serial.print(current_impulse_angle, 1);
Serial.println("°");
    Serial.print("Velocidad impulso: "); Serial.print(SWING_IMPULSE_SPEED, 1);
Serial.println(" steps/sec");
}

void run_swing_up() {
    float elapsed_time = (current_time - swing_start_time) / 1000.0;
    float phase_elapsed = (current_time - swing_phase_start_time) / 1000.0;

    // Debug periódico
    if (current_time - last_debug_print > DEBUG_INTERVAL) {
        Serial.print("SWING DEBUG - Tiempo: "); Serial.print(elapsed_time, 1);
        Serial.print("s, θ: "); Serial.print(theta, 1);
        Serial.print("°, θ_max: "); Serial.print(theta_max_achieved, 1);
        Serial.print("°, Estado: ");
        switch(swing_state) {
            case SWING_IMPULSE: Serial.print("IMPULSO"); break;
            case SWING_WAIT: Serial.print("ESPERA"); break;
            case SWING_DIRECTION_CHANGE: Serial.print("CAMBIO_DIR"); break;
        }
        Serial.print(", Dir: "); Serial.print(swing_direction ? "+" : "-");
        Serial.print(", Energía: "); Serial.println(pendulum_energy, 2);
        last_debug_print = current_time;
    }

    // VERIFICAR SI SE ALCANZÓ LA POSICIÓN INVERTIDA
    if (is_pendulum_upright()) {
```

```
        if (!upright_achieved) {
            // Parar stepper inmediatamente
            stepper->hard_stop();

            // Guardar posición de referencia del stepper
            phi_reference = phi;
            upright_achieved = true;

            Serial.println("*** ¡POSICIÓN INVERTIDA ALCANZADA! ***");
            Serial.print("Tiempo transcurrido: "); Serial.print(elapsed_time, 1);
Serial.println("s");
            Serial.print("Ángulo final péndulo: "); Serial.print(theta, 1);
Serial.println("°");
            Serial.print("Posición stepper guardada: ");
Serial.print(phi_reference, 1); Serial.println("°");
        }

        // TRANSICIÓN AL CONTROL
        system_state = STATE_CONTROL;

        // Reconfigurar stepper para control (velocidades moderadas)
        stepper->set_max_speed(200);
        stepper->set_acceleration(500);

        // Reset controller states
        pid_integral = 0.0;
        pid_error_prev = 0.0;
        integral_error = 0.0;

        Serial.println("=== INICIANDO CONTROL DE ESTABILIZACIÓN ===");
        return;
    }

    // Verificar timeout
    if (elapsed_time > SWING_MAX_DURATION || swing_cycle_count >=
SWING_MAX_CYCLES) {
        Serial.println("*** TIMEOUT EN SWING-UP ***");
        Serial.print("Tiempo: "); Serial.print(elapsed_time, 1);
Serial.println("s");
        Serial.print("Ciclos: "); Serial.println(swing_cycle_count);
        Serial.print("Máximo ángulo alcanzado: ");
Serial.print(theta_max_achieved, 1); Serial.println("°");

        // Pasar al control con la mejor posición alcanzada
        stepper->hard_stop();
```

```
        phi_reference = phi;
        system_state = STATE_CONTROL;

        // Reconfigurar stepper para control
        stepper->set_max_speed(200);
        stepper->set_acceleration(500);
        return;
    }

    // MÁQUINA DE ESTADOS DEL SWING-UP
    switch(swing_state) {
        case SWING_IMPULSE:
            // Ejecutar impulso si el stepper no se está moviendo
            if (stepper->get_device_state() == INACTIVE) {
                float impulse_angle = swing_direction ? current_impulse_angle : -
current_impulse_angle;
                move_stepper_fast(impulse_angle, SWING_IMPULSE_SPEED);

                Serial.print("Ejecutando impulso: "); Serial.print(impulse_angle,
1);
                Serial.print("° (Ciclo "); Serial.print(swing_cycle_count + 1);
Serial.println(")");

                swing_state = SWING_WAIT;
                swing_phase_start_time = current_time;
            }
            break;

        case SWING_WAIT:
            // Esperar durante la oscilación del péndulo
            if (phase_elapsed >= SWING_WAIT_TIME) {
                swing_state = SWING_DIRECTION_CHANGE;
            }
            break;

        case SWING_DIRECTION_CHANGE:
            // Cambiar dirección y preparar siguiente impulso
            swing_direction = !swing_direction;
            swing_cycle_count++;

            // Ajuste automático de la amplitud basado en el progreso
            if (swing_cycle_count % 5 == 0) { // Cada 5 ciclos evaluar progreso
                if (theta_max_achieved < 45.0) {
                    // Péndulo no está ganando altura, incrementar impulso
```

```
                    current_impulse_angle = min(current_impulse_angle + 5.0,
60.0);
                    Serial.print("Incrementando impulso a: ");
Serial.print(current_impulse_angle, 1); Serial.println("°");
                } else if (theta_max_achieved > 150.0) {
                    // Péndulo cerca del objetivo, reducir impulso para control
fino
                    current_impulse_angle = max(current_impulse_angle - 2.0,
10.0);
                    Serial.print("Reduciendo impulso a: ");
Serial.print(current_impulse_angle, 1); Serial.println("°");
                }
            }

            swing_state = SWING_IMPULSE;
            swing_phase_start_time = current_time;
            break;
    }
}

/*****************************************************************************
* Main control state machine
*/
void run_control_loop() {
    // Only run control loop at specified intervals
    if (millis() - control_loop_timer < CONTROL_INTERVAL) {
        return;
    }
    control_loop_timer = millis();

    update_states();

    float control_output = 0.0;

    switch (system_state) {
        case STATE_SWING_UP:
            run_swing_up();
            control_output = current_impulse_angle * (swing_direction ? 1.0 : -
1.0); // Para monitoreo
            break;

        case STATE_CONTROL:
            {
                // CONTROL ACTIVO CON STEPPER
                if (current_controller == CONTROLLER_PID) {
```

```
                control_output = compute_pid_control();
            } else {
                control_output = compute_lqr_control();
            }

            // Aplicar control como corrección de posición
            float target_phi = phi_reference + control_output;
            float phi_error = target_phi - phi;

            // Mover stepper solo si el error es significativo
            if (abs(phi_error) > 1.0 && stepper->get_device_state() ==
INACTIVE) {
                move_stepper_by(phi_error * 0.5); // Movimiento suave
            }
        }
        break;

    default:
        control_output = 0.0;
        break;
    }
}

/*******************************************************************************
* Setup and main loop
*/
void setup() {
    // Configure pins
    pinMode(LED_PIN, OUTPUT);
    pinMode(D4, INPUT_PULLUP);
    pinMode(D5, INPUT_PULLUP);

    // Initialize communication
    Serial.begin(BAUD_RATE);
    Serial.println("=== SISTEMA DE CONTROL PÉNDULO - SWING-UP CON STEPPER ===");
    ctrl.init(Serial, CTRL_DEBUG);

    // Configure encoder
    encoder = new RotaryEncoder(ENC_A_PIN, ENC_B_PIN,
RotaryEncoder::LatchMode::TWO03);
    attachInterrupt(digitalPinToInterrupt(ENC_A_PIN), encoderISR, CHANGE);
    attachInterrupt(digitalPinToInterrupt(ENC_B_PIN), encoderISR, CHANGE);
    Serial.println("Encoder del péndulo configurado (sensor únicamente)");

    // Initialize stepper
```

```
    Serial.println("Inicializando motor stepper (swing-up y 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: No se pudo inicializar el driver del stepper");
        while(1);
    }

    stepper->attach_flag_irq(&stepperISR);
    stepper->enable_flag_irq();
    stepper->set_home();

    // Configuración inicial para movimientos normales
    stepper->set_max_speed(200);
    stepper->set_acceleration(500);

    Serial.println("Motor stepper inicializado correctamente");

    // Initialize timing
    prev_time = millis();
    control_loop_timer = millis();
    last_debug_print = millis();

    Serial.println("=== SISTEMA COMPLETAMENTE INICIALIZADO ===");
    Serial.println("Controlador de Péndulo Invertido - Swing-up con Stepper");
    Serial.println("Encoder: Sensor de ángulo del péndulo");
    Serial.println("Stepper: Swing-up (impulsos rápidos) y Control (movimientos
finos)");
    Serial.println("=== ESPERANDO COMANDOS ===");
}

void loop() {
    int command;
    float action[NUM_ACTIONS];
    float observation[NUM_OBS];
    ControlComms::StatusCode rx_code;

    // Always run control loop if active
    if (system_state == STATE_SWING_UP || system_state == STATE_CONTROL) {
        run_control_loop();
    }

    // Handle communication
    rx_code = ctrl.receive_action<NUM_ACTIONS>(&command, action);
```

```
    if (rx_code == ControlComms::OK) {
        switch (command) {
            case CMD_SET_HOME:
                set_stepper_home();
                Serial.println("Home establecido");
                break;

            case CMD_MOVE_TO:
                if (system_state == STATE_IDLE) {
                    move_stepper_to(action[0]);
                    Serial.print("Moviendo stepper a: "); Serial.print(action[0],
1); Serial.println("°");
                }
                break;

            case CMD_MOVE_BY:
                if (system_state == STATE_IDLE) {
                    move_stepper_by(action[0]);
                    Serial.print("Moviendo stepper por: ");
Serial.print(action[0], 1); Serial.println("°");
                }
                break;

            case CMD_SET_STEP_MODE:
                set_step_mode((int)action[0]);
                set_stepper_home();
                Serial.print("Modo de paso establecido: ");
Serial.println((int)action[0]);
                break;

            case CMD_SELECT_CONTROLLER:
                current_controller = (ControllerType)((int)action[0]);
                Serial.print("Controlador seleccionado: ");
                Serial.println(current_controller == CONTROLLER_PID ? "PID" :
"LQR");
                break;

            case CMD_SET_PID_GAINS:
                K_p = action[0];
                K_i = action[1];
                K_d = action[2];
                Serial.print("Ganancias PID - Kp: "); Serial.print(K_p, 6);
                Serial.print(", Ki: "); Serial.print(K_i, 6);
                Serial.print(", Kd: "); Serial.println(K_d, 6);
```

```
                break;

            case CMD_SET_LQR_GAINS:
                for (int i = 0; i < 5; i++) {
                    K_lqr[i] = action[i];
                }
                Serial.print("Ganancias LQR: [");
                for (int i = 0; i < 5; i++) {
                    Serial.print(K_lqr[i], 6);
                    if (i < 4) Serial.print(", ");
                }
                Serial.println("]");
                break;

            case CMD_START_CONTROL:
                if (system_state == STATE_IDLE) {
                    system_state = STATE_SWING_UP;
                    reset_swing_up_variables();

                    // Reset controller states
                    pid_integral = 0.0;
                    pid_error_prev = 0.0;
                    integral_error = 0.0;

                    Serial.println("*** CONTROL INICIADO ***");
                    Serial.println("Fase 1: Swing-up con impulsos del motor
stepper");
                    Serial.println("Fase 2: Control automático al alcanzar
180°");
                } else {
                    Serial.println("Error: Sistema no está en estado IDLE");
                }
                break;

            case CMD_STOP_CONTROL:
                system_state = STATE_IDLE;
                stepper->hard_stop(); // Detener stepper inmediatamente
                Serial.println("*** CONTROL DETENIDO ***");
                break;

            default:
                Serial.print("Comando desconocido: "); Serial.println(command);
                break;
        }
```

```
        // Always update states and send observation
        update_states();

        // Prepare observation
        observation[0] = theta; // Pendulum angle
        observation[1] = phi; // Rotor angle
        observation[2] = dtheta; // Pendulum velocity
        observation[3] = dphi; // Rotor velocity
        observation[4] = (system_state == STATE_SWING_UP) ?
current_impulse_angle : 0.0; // Swing impulse angle
        observation[5] = (system_state == STATE_SWING_UP) ? pendulum_energy :
integral_error; // Energy or control error

        // Determine status
        int status = STATUS_OK;
        if (system_state == STATE_SWING_UP) {
            status = STATUS_SWING_UP;
        } else if (system_state == STATE_CONTROL) {
            status = is_pendulum_upright() ? STATUS_UPRIGHT_ACHIEVED :
STATUS_CONTROL_ACTIVE;
        } else if (stepper->get_device_state() != INACTIVE) {
            status = STATUS_STP_MOVING;
        }

        // Send observation
        ctrl.send_observation(status, millis(), false, observation, NUM_OBS);

    } else if (rx_code == ControlComms::ERROR) {
        // Solo reportar errores críticos, no timeouts normales
        if (millis() % 10000 == 0) { // Cada 10 segundos
            Serial.println("Estado del sistema: Esperando comandos...");
        }
    }

    // LED de estado para debugging visual
    static unsigned long last_led_update = 0;
    if (millis() - last_led_update > 1000) {
        digitalWrite(LED_PIN, !digitalRead(LED_PIN)); // Toggle LED cada segundo
        last_led_update = millis();
    }
}
```