INFORME TRABAJO PRÁCTICO FINAL
ELECTRÓNICA DIGITAL 3

“Seguidor de Luz”

INTEGRANTES

Álvarez, Agustín
Cáceres, Juan Manuel

COMISIÓN

Gallardo, Fernando

1. Resumen Ejecutivo

El presente proyecto consiste en el desarrollo de un sistema seguidor de luz basado en la placa de desarrollo LPC1769. El sistema utiliza cuatro sensores LDR para detectar la dirección de mayor iluminación y orientar un conjunto de dos motores DC en dos ejes mediante un algoritmo de control PID. Además, integra múltiples periféricos del microcontrolador: lectura analógica mediante ADC en modo burst, generación de señales PWM por software utilizando SysTick, salida analógica por DAC controlada mediante GPDMA, manejo de interrupciones externas (EINT) con antirrebote por Timer0, y un modo alternativo de control manual mediante UART, simulando un joystick.

El objetivo del trabajo es aplicar los conocimientos adquiridos en programación de microcontroladores, integrando varios módulos de la LPC1769 dentro de un sistema funcional, estable y orientado a aplicaciones mecatrónicas.

2. Introducción

Los sistemas seguidores de luz (light followers) son ampliamente utilizados en robótica para orientar dispositivos hacia una fuente luminosa, como paneles solares, antenas o sistemas de puntería. El objetivo de este trabajo fue diseñar e implementar un sistema capaz de detectar la posición de mayor iluminación y orientar un actuador en consecuencia, utilizando exclusivamente recursos de la LPC1769.

El proyecto integra múltiples periféricos y conceptos clave vistos en la materia Digital 3 y Sistemas de Control:

- Conversión analógica/digital.
- Interrupciones internas y externas.
- Timers y SysTick.
- Control PWM por software.
- Control PID.
- Comunicación serial.
- Aplicación de DMA.

El resultado final es un sistema robusto, que opera tanto de manera autónoma como manual, y permite analizar la señal del PID en un osciloscopio mediante el DAC.

3. Descripción General del Sistema

3.1 Arquitectura General

El sistema se organiza en tres bloques principales:

Adquisición de señales:

- Cuatro LDR distribuidos en forma de domo.
- Módulo ADC en modo burst, obteniendo lecturas de los sensores de manera continua y escribiendo continuamente en las variables de cada canal respectivamente.

Procesamiento:

- Dos controladores PID independientes (uno por eje).
- Cálculo de errores en función de las diferencias medidas entre los LDRs.
- Máquina de estados según el modo de funcionamiento: LDR vs Joystick.

Actuación:

- Dos motores de DC, cada uno con indicador de dirección (GPIO) y PWM generado por software utilizando pulsos de SysTick.
- Señal PID enviada al DAC (a través de GPDMA) vista por un instrumento de medición osciloscopio.

3.2 Flujo General del Programa

El programa combina tareas ejecutadas por interrupciones y procesamientos continuos en el bucle principal.

Las interrupciones manejan las funciones críticas: SysTick genera el PWM por software y actualiza el modo joystick, Timer0 aplica antirrebote a los botones, Timer1 sincroniza la actualización del DAC mediante DMA, y las interrupciones externas EINT0, EINT1 y EINT2 cambian estados y modos de operación. La UART recibe comandos del joystick.

En el bucle principal se encarga de procesar las lecturas del ADC, ejecutar los controladores PID, actualizar el DAC y ajustar la velocidad y dirección de los motores según el resultado del control.

4. Hardware

4.1 Sensores LDR

Se emplean cuatro LDR distribuidos en un domo para obtener diferencias de luz entre los ejes X e Y. Cada sensor está conectado a una entrada ADC (canales AD0.0, AD0.1, AD0.2 y AD0.5).

El ADC opera en modo burst, permitiendo muestreos continuos sin intervención del procesador y así manteniendo en las variables de valor de cada LDR el valor más actual posible que será usado luego para el cálculo del error y la posterior compensación PID.

4.2 Motores DC

Se utilizan dos motores, uno para cada eje.

Cada motor cuenta con:

- Pines de GPIO para determinar el sentido de giro (2 bits de control por cada motor [1,0] o [0,1]).
- Señal PWM implementada mediante pulsos con SysTick.

El duty cycle del PWM se regula entre 0 y 100 ticks, equivalente a un período de 10 ms de duración, que es una duración estándar para un control PWM certero.

4.3 Botones y EINT

El sistema cuenta con tres botones pulsadores configurados como interrupciones externas:

Botón	***Interrupción***	***Función***
Botón 0	EINT0	Habilita o deshabilita los motores
Botón 1	EINT1	Cambia entre los errores de cada PID a visualizar por el DAC

Botón 2	EINT2	Cambia modo entre lecturas de LDR o joystick

Los botones implementan la función de antirrebote por medio de Timer0, donde al ser presionados, se inicia un contador en el cual cualquier otra pulsación que se detecte será descartada.

4.4 DAC + Osciloscopio

El error del PID seleccionado para ser visualizado se envía al DAC.

Timer1 activa periódicamente la copia del valor al DAC usando un canal GPDMA, reduciendo carga del CPU y permitiendo una señal estable para medir.

4.5 UART — Modo Joystick

El sistema también puede ser controlado manualmente mediante UART, recibiendo los comandos desde otro dispositivo que se comunique mediante este protocolo (en el caso aplicado es mediante un microcontrolador ESP-WROOM-32 y una computadora, que por consola se envían los comandos al ESP-WROOM-32 mediante USB y éste a la LPC1769 mediante UART):

0 = detención
1 = izquierda
2 = derecha
3 = abajo
4 = arriba

Estos comandos generan LDRs virtuales, reemplazando el control por luz real. Donde para implementar esta funcionalidad lo que se hace es que se carga un valor de error determinado en vez de ser calculado mediante la diferencia de las mediciones LDR.

5. Software

5.1 Periféricos configurados

Función	***Periférico***	***Descripción***
configADC()	ADC	Lectura continua de los 4 LDR
configGPIO()	GPIO	Dirección y PWM de motores
configSysTick()	SysTick	Interrupción cada 100 µs, PWM y joystick
configTimer()	Timer1	Antirrebote y actualización del DAC
configUART()	UART0	Recepción de comandos del joystick
configDAC()	DAC	Salida analógica del error PID
configGPDMA()	DMA	Transferencia eficiente al DAC

5.2 Implementaciones

En el software, mediante funciones individuales, se configuran los periféricos necesarios para el funcionamiento del sistema. (Funciones inicializadas en “config”, seguidas por el periférico a configurar).

El control se basa en dos PID independientes, uno por eje, que calculan el error a partir de las diferencias entre los sensores y determinan tanto la dirección como el duty cycle de los motores.

SysTick implementa el PWM por software comparando un contador interno con los duty cycles calculados. El Timer1 asegura una actualización estable del DAC a través del DMA, y la UART mantiene un buffer circular para procesar los comandos recibidos.

De esta forma, el software integra el procesamiento del PID, la lectura del ADC, el control de motores y el manejo de interrupciones para lograr un seguimiento estable y continuo.

6. Resultados

6.1 Pruebas del Seguidor de Luz

El sistema seguidor de luz fue evaluado bajo distintas condiciones de iluminación y mostró un comportamiento estable, orientándose rápidamente hacia la mayor fuente de luz sin generar oscilaciones notables.

El análisis de la señal del PID, visualizado a través del DAC en un osciloscopio, evidenció transitorios suaves y una respuesta estable incluso ante variaciones bruscas en las mediciones.

El modo joystick operó correctamente los comandos enviados mediante UART, que permitieron movimientos precisos y con baja latencia.

La conmutación entre modos se realizó de manera inmediata gracias al manejo de interrupciones externas y sus antirrebotes.

En conjunto, los resultados confirmaron que el sistema mantiene un desempeño confiable tanto en funcionamiento automático como en control manual.

7. Conclusiones

El proyecto permitió integrar de manera efectiva diversos periféricos de la LPC1769 dentro de un mismo sistema, logrando un seguidor de luz estable y versátil.

La implementación de control PID, la generación de PWM por software, el uso del ADC en modo burst, la gestión de interrupciones y la aplicación de DMA fueron herramientas eficaces para cumplir los objetivos del sistema.

El sistema funcionó correctamente tanto en modo automático como en modo manual y ofreció una herramienta clara de depuración mediante la salida analógica del DAC.

Como trabajo futuro se podría:

- Utilizar PWM por hardware para mayor resolución
- Implementar filtros digitales para el procesamiento de señales del ADC
- Mejorar el sistema mecánico y aumentar precisión

8. Anexo — Código Completo

```
/*
 * Copyright 2022 NXP
 * NXP confidential.
 * This software is owned or controlled by NXP and may only be used strictly
 * in accordance with the applicable license terms.  By expressly accepting
 * such terms or by downloading, installing, activating and/or otherwise using
 * the software, you are agreeing that you have read, and that you agree to
 * comply with and are bound by, such license terms.  If you do not agree to
 * be bound by the applicable license terms, then you may not retain, install,
 * activate or otherwise use the software.
 */

#ifdef __USE_CMSIS
#include "LPC17xx.h"
#endif

#include <stdlib.h>
#include <string.h>
#include <math.h>

#include <cr_section_macros.h>

#include "lpc17xx_dac.h"
#include "lpc17xx_gpdma.h"
#include "lpc17xx_timer.h"
#include "lpc17xx_uart.h"

// Macro functions
#define max(a, b) ((a) > (b) ? (a) : (b))
#define min(a, b) ((a) < (b) ? (a) : (b))
#define constrain(x, low, high) (((x) < (low)) ? (low) : (((x) > (high)) ? (high) : (x)))

// GPDMA constants
#define GPDMA_CHANNEL_0 0

// SysTick & Timer constants
#define SYSTICK_TIME_IN_US 100 // 0.1[ms]
#define TIMER_TIME_IN_US 10000 // 10[ms]
#define PWM_CYCLES 100 // Number of SYSTICK_TIME_IN_US to consider a PWM cycle (100 * 0.1[ms] = 10[ms])
#define JOYSTICK_CYCLES 1000 // Number of SYSTICK_TIME_IN_US to achieve a movement in joystick mode (1000 * 0.1[ms] =
100[ms])
#define MATCH_VALUE_DEBOUNCE 20 // Number of TIMER_TIME_IN_US to achieve debounce (20 * 10[ms] = 200[ms])
#define MATCH_VALUE_DAC 1 // Number of TIMER_TIME_IN_US to achieve DAC update rate (1 * 10[ms] = 10[ms])

// General constants
#define MAX_THROTTLE 64 // Maximum throttle level
#define JOYSTICK_BUFFER_SIZE 1024 // Size of the joystick buffer

// PID 0 constants
#define KP_0 0.03 // Proportional gain for the control algorithm
#define KI_0 0.00035 // Integral gain for the control algorithm
#define KD_0 0.000015 // Derivative gain for the control algorithm
#define WINDUP_LIMIT_0 1000 // Integral windup limit for Motor 0

// PID 1 constants
#define KP_1 0.03 // Proportional gain for the control algorithm
#define KI_1 0.00035 // Integral gain for the control algorithm
#define KD_1 0.000015 // Derivative gain for the control algorithm
#define WINDUP_LIMIT_1 1000 // Integral windup limit for Motor 1
```

```c
// ADC variables
int static LDRValue_0;
int static LDRValue_1;
int static LDRValue_2;
int static LDRValue_3;

// DAC variables
uint32_t static DACValue = 0; // Value to output via DAC
int static DACUpdateFlag = 0; // Flag to indicate when to update the DAC output
int static errorSelection = 0; // Variable to select which error to output via DAC

// EINT variables
int static debounceFlag = 0;

// GPDMA variables
// (none)

// UART variables
volatile uint8_t rx_data = 0; // Variable to store received UART data

// General variables
int static modeSelection = 0; // 0=LDRs mode, 1=joystick mode
int static joystickBuffer[JOYSTICK_BUFFER_SIZE]; // List of characters received by UART
int static joystickCounter = 0; // Counter for joystick cycles
int static joystickIndex = 0; // Current reading index in joystickBuffer
int static joystickSize = 0; // Number of entries stored in joystickBuffer

// Motor 0 variables
int static motorEnable_0 = 0; // Enable state of Motor 0 (0=off, 1=on)
int static motorDirection_0 = 0; // Direction of Motor 0
int static motorThrottle_0 = 0; // Throttle level of Motor 0
int static pwmCounter_0 = 0; // Counter for PWM cycles of Motor 0

// Motor 1 variables
int static motorEnable_1 = 0; // Enable state of Motor 1 (0=off, 1=on)
int static motorDirection_1 = 0; // Direction of Motor 1
int static motorThrottle_1 = 0; // Throttle level of Motor 1
int static pwmCounter_1 = 0; // Counter for PWM cycles of Motor 1

// PID 0 variables
double static setpoint_0 = 0; // Desired value for Motor 0 control
double static measuredValue_0 = 0; // Measured value for Motor 0 control
double static error_0 = 0; // Current error for Motor 0 control
double static previousError_0 = 0; // Previous error for Motor 0 control
double static integral_0 = 0; // Integral term for Motor 0 control
double static derivative_0 = 0; // Derivative term for Motor 0 control
double static output_0 = 0; // PID output for Motor 0 control

// PID 1 variables
double static setpoint_1 = 0; // Desired value for Motor 1 control
double static measuredValue_1 = 0; // Measured value for Motor 1 control
double static error_1 = 0; // Current error for Motor 1 control
double static previousError_1 = 0; // Previous error for Motor 1 control
double static integral_1 = 0; // Integral term for Motor 1 control
double static derivative_1 = 0; // Derivative term for Motor 1 control
double static output_1 = 0; // PID output for Motor 1 control

void configADC();
void configDAC();
void configEINT();
void configGPDMA();
void configGPIO();
```

```c
void configSysTick();
void configTimer();
void configUART();

void UARTSendString(uint8_t *str);
void processThrottleAndDirection();
int calculatePID_0();
int calculatePID_1();
void updateMotor0();
void updateMotor1();
void updateDAC();

int main() {
    SystemInit();
    configADC();
    configDAC();
    configEINT();
    configGPDMA();
    configGPIO();
    configSysTick();
    configTimer();
    configUART();
    while (1) {
        switch (modeSelection) {
            case 0: // LDRs mode
                processThrottleAndDirection();
                updateMotor0();
                updateMotor1();
                break;
            case 1: // Joystick mode
                switch (joystickBuffer[joystickIndex]) {
                    case 0: // No movement
                        LDRValue_0 = 0;
                        LDRValue_1 = 0;
                        LDRValue_2 = 0;
                        LDRValue_3 = 0;
                        break;
                    case 1: // Right
                        LDRValue_0 = 1024;
                        LDRValue_1 = 0;
                        LDRValue_2 = 0;
                        LDRValue_3 = 0;
                        break;
                    case 2: // Left
                        LDRValue_0 = 0;
                        LDRValue_1 = 1024;
                        LDRValue_2 = 0;
                        LDRValue_3 = 0;
                        break;
                    case 3: // Up
                        LDRValue_0 = 0;
                        LDRValue_1 = 0;
                        LDRValue_2 = 1024;
                        LDRValue_3 = 0;
                        break;
                    case 4: // Down
                        LDRValue_0 = 0;
                        LDRValue_1 = 0;
                        LDRValue_2 = 0;
                        LDRValue_3 = 1024;
                        break;
                    default: // Error
                        LDRValue_0 = 0;
```

```
                        LDRValue_1 = 0;
                        LDRValue_2 = 0;
                        LDRValue_3 = 0;
                        break;
                }
                processThrottleAndDirection();
                updateMotor0();
                updateMotor1();
                break;
            default:
                break;
        }
        if (DACUpdateFlag == 1) {
            updateDAC();
            configGPDMA();
            DACUpdateFlag = 0;
        }
    }
    return 0;
}

/*
 * CONFIGURATION METHODS
 */

void configADC() {
    LPC_PINCON->PINSEL1 &= ~(3 << 14); // Clear P0.23 function bits
    LPC_PINCON->PINSEL1 |= (1 << 14); // Set P0.23 as AD0.0
    LPC_PINCON->PINSEL1 &= ~(3 << 16); // Clear P0.24 function bits
    LPC_PINCON->PINSEL1 |= (1 << 16); // Set P0.24 as AD0.1
    LPC_PINCON->PINSEL1 &= ~(3 << 18); // Clear P0.25 function bits
    LPC_PINCON->PINSEL1 |= (1 << 18); // Set P0.25 as AD0.2
    LPC_PINCON->PINSEL3 &= ~(3 << 30); // Clear P1.31 function bits
    LPC_PINCON->PINSEL3 |= (3 << 30); // Set P1.31 as AD0.5

    LPC_SC->PCONP |= (1 << 12); // Power up ADC
    LPC_SC->PCLKSEL0 &= ~(3 << 24); // Clear PCLK_ADC
    LPC_SC->PCLKSEL0 |= (3 << 24); // Set PCLK_ADC to CCLK/8
    LPC_ADC->ADCR = (1 << 0) | (1 << 1) | (1 << 2) | (1 << 5) | (124 << 8) | (0 << 16) | (1 << 21);
    LPC_ADC->ADINTEN = (1 << 0) | (1 << 1) | (1 << 2) | (1 << 5); // Enable interrupts for channels 0, 1, 2 and 5
    NVIC_EnableIRQ(ADC_IRQn);
    LPC_ADC->ADCR |= (1 << 16); // Start burst conversion
}

void configDAC() {
    LPC_PINCON->PINSEL1 &= ~(3 << 20); // Clear P0.26 function bits
    LPC_PINCON->PINSEL1 |= (2 << 20); // Set P0.26 as AOUT
    LPC_PINCON->PINMODE1 |= (3 << 20); // Set P0.26 with PULL_DOWN

    LPC_DAC->DACR |= (1 << 16); // Set bias in 400kHz
    LPC_DAC->DACCTRL |= (12 << 0); // Enable DMA & timeout
    LPC_DAC->DACCNTVAL |= 250000;

    LPC_DAC->DACR &= ~(1023 << 6); // Set the P0.26 DAC output in LOW
}

void configEINT() {
    LPC_PINCON->PINSEL4 &= ~(3 << 20); // Clear P2.10 function bits
    LPC_PINCON->PINSEL4 |= (1 << 20); // Set P2.10 as EINT0
    LPC_PINCON->PINMODE4 &= ~(3 << 20); // Set P2.10 with PULL-UP
    LPC_GPIO2->FIODIR &= ~(1 << 10); // Set P2.10 as INPUT
    LPC_SC->EXTINT |= (1 << 0); // Set EINT0 as external interrupt
    LPC_SC->EXTMODE |= (1 << 0); // Set EINT0 as edge sensitive
```

```
    LPC_SC->EXTPOLAR &= ~(1 << 0); // Set EINT0 interruption in falling edge
    NVIC_EnableIRQ(EINT0_IRQn);

    LPC_PINCON->PINSEL4 &= ~(3 << 22); // Clear P2.11 function bits
    LPC_PINCON->PINSEL4 |= (1 << 22); // Set P2.11 as EINT1
    LPC_PINCON->PINMODE4 &= ~(3 << 22); // Set P2.11 with PULL-UP
    LPC_GPIO2->FIODIR &= ~(1 << 11); // Set P2.11 as INPUT
    LPC_SC->EXTINT |= (1 << 1); // Set EINT1 as external interrupt
    LPC_SC->EXTMODE |= (1 << 1); // Set EINT1 as edge sensitive
    LPC_SC->EXTPOLAR &= ~(1 << 1); // Set EINT1 interruption in falling edge
    NVIC_EnableIRQ(EINT1_IRQn);

    LPC_PINCON->PINSEL4 &= ~(3 << 24); // Clear P2.12 function bits
    LPC_PINCON->PINSEL4 |= (1 << 24); // Set P2.12 as EINT2
    LPC_PINCON->PINMODE4 &= ~(3 << 24); // Set P2.12 with PULL-UP
    LPC_GPIO2->FIODIR &= ~(1 << 12); // Set P2.12 as INPUT
    LPC_SC->EXTINT |= (1 << 2); // Set EINT2 as external interrupt
    LPC_SC->EXTMODE |= (1 << 2); // Set EINT2 as edge sensitive
    LPC_SC->EXTPOLAR &= ~(1 << 2); // Set EINT2 interruption in falling edge
    NVIC_EnableIRQ(EINT2_IRQn);

    LPC_PINCON->PINSEL4 &= ~(3 << 26); // Clear P2.13 function bits
    LPC_PINCON->PINSEL4 |= (1 << 26); // Set P2.13 as EINT3
    LPC_PINCON->PINMODE4 &= ~(3 << 26); // Set P2.13 with PULL-UP
    LPC_GPIO2->FIODIR &= ~(1 << 13); // Set P2.13 as INPUT
    LPC_SC->EXTINT |= (1 << 3); // Set EINT3 as external interrupt
    LPC_SC->EXTMODE |= (1 << 3); // Set EINT3 as edge sensitive
    LPC_SC->EXTPOLAR &= ~(1 << 3); // Set EINT3 interruption in falling edge
    NVIC_EnableIRQ(EINT3_IRQn);
}

void configGPDMA() {
    GPDMA_Init();
    GPDMA_Channel_CFG_Type GPDMA;
    GPDMA.ChannelNum = GPDMA_CHANNEL_0;
    GPDMA.TransferSize = 1; // Single word
    GPDMA.TransferWidth = GPDMA_WIDTH_WORD;
    GPDMA.SrcMemAddr = (uint32_t)DACValue;
    GPDMA.DstMemAddr = 0; // Destination peripheral address
    GPDMA.TransferType = GPDMA_TRANSFERTYPE_M2P;
    GPDMA.SrcConn = 0; // Memory
    GPDMA.DstConn = GPDMA_CONN_DAC; // DAC peripheral connection
    GPDMA.DMALLI = 0;
    GPDMA_Setup(&GPDMA);
    //NVIC_EnableIRQ(DMA_IRQn);
}

void configGPIO() {
    LPC_PINCON->PINSEL4 &= ~(3 << 0); // Set P2.0 as GPIO
    LPC_PINCON->PINMODE4 &= ~(1 << 0); // Clear P2.0 mode bits
    LPC_PINCON->PINMODE4 |= (2 << 0); // Set P2.0 neither PULL-UP nor PULL-DOWN
    LPC_GPIO2->FIODIR |= (1 << 0); // Set P2.0 as OUTPUT

    LPC_PINCON->PINSEL4 &= ~(3 << 2); // Set P2.1 as GPIO
    LPC_PINCON->PINMODE4 &= ~(1 << 2); // Clear P2.1 mode bits
    LPC_PINCON->PINMODE4 |= (2 << 2); // Set P2.1 neither PULL-UP nor PULL-DOWN
    LPC_GPIO2->FIODIR |= (1 << 1); // Set P2.1 as OUTPUT

    LPC_PINCON->PINSEL4 &= ~(3 << 4); // Set P2.2 as GPIO
    LPC_PINCON->PINMODE4 &= ~(1 << 4); // Clear P2.2 mode bits
    LPC_PINCON->PINMODE4 |= (2 << 4); // Set P2.2 neither PULL-UP nor PULL-DOWN
    LPC_GPIO2->FIODIR |= (1 << 2); // Set P2.2 as OUTPUT
```

```
    LPC_PINCON->PINSEL4 &= ~(3 << 6); // Set P2.3 as GPIO
    LPC_PINCON->PINMODE4 &= ~(1 << 6); // Clear P2.3 mode bits
    LPC_PINCON->PINMODE4 |= (2 << 6); // Set P2.3 neither PULL-UP nor PULL-DOWN
    LPC_GPIO2->FIODIR |= (1 << 3); // Set P2.3 as OUTPUT

    LPC_PINCON->PINSEL4 &= ~(3 << 8); // Set P2.4 as GPIO
    LPC_PINCON->PINMODE4 &= ~(1 << 8); // Clear P2.4 mode bits
    LPC_PINCON->PINMODE4 |= (2 << 8); // Set P2.4 neither PULL-UP nor PULL-DOWN
    LPC_GPIO2->FIODIR |= (1 << 4); // Set P2.4 as OUTPUT

    LPC_PINCON->PINSEL4 &= ~(3 << 10); // Set P2.5 as GPIO
    LPC_PINCON->PINMODE4 &= ~(1 << 10); // Clear P2.5 mode bits
    LPC_PINCON->PINMODE4 |= (2 << 10); // Set P2.5 neither PULL-UP nor PULL-DOWN
    LPC_GPIO2->FIODIR |= (1 << 5); // Set P2.5 as OUTPUT

    LPC_GPIO2->FIOMASK |= ~0x3F; // Mask all pins except P2.0 to P2.5

    LPC_GPIO2->FIOCLR |= (1 << 0); // Set P2.0 in LOW
    LPC_GPIO2->FIOCLR |= (1 << 1); // Set P2.1 in LOW
    LPC_GPIO2->FIOCLR |= (1 << 2); // Set P2.2 in LOW
    LPC_GPIO2->FIOCLR |= (1 << 3); // Set P2.3 in LOW
    LPC_GPIO2->FIOCLR |= (1 << 4); // Set P2.4 in LOW
    LPC_GPIO2->FIOCLR |= (1 << 5); // Set P2.5 in LOW
}

void configSysTick() {
    SysTick->LOAD = (SystemCoreClock / 1000000) * SYSTICK_TIME_IN_US - 1;
    SysTick->VAL = 0;
     SysTick->CTRL = (1 << 0) | (1 << 1) | (1 << 2); // Enable SysTick counter, enable SysTick interruptions and
select internal clock
}

void configTimer() {
    TIM_TIMERCFG_Type timer;
    timer.PrescaleOption = TIM_PRESCALE_USVAL;
    timer.PrescaleValue = TIMER_TIME_IN_US;
    TIM_Init(LPC_TIM0, TIM_TIMER_MODE, &timer);
    TIM_Init(LPC_TIM1, TIM_TIMER_MODE, &timer);

    TIM_MATCHCFG_Type matchDebounce;
    matchDebounce.MatchChannel = 0;
    matchDebounce.IntOnMatch = ENABLE;
    matchDebounce.StopOnMatch = ENABLE;
    matchDebounce.ResetOnMatch = ENABLE;
    matchDebounce.ExtMatchOutputType = TIM_EXTMATCH_NOTHING;
    matchDebounce.MatchValue = MATCH_VALUE_DEBOUNCE;
    TIM_ConfigMatch(LPC_TIM0, &matchDebounce);

    TIM_MATCHCFG_Type matchDAC;
    matchDAC.MatchChannel = 0;
    matchDAC.IntOnMatch = ENABLE;
    matchDAC.StopOnMatch = DISABLE;
    matchDAC.ResetOnMatch = ENABLE;
    matchDAC.ExtMatchOutputType = TIM_EXTMATCH_NOTHING;
    matchDAC.MatchValue = MATCH_VALUE_DAC;
    TIM_ConfigMatch(LPC_TIM1, &matchDAC);

    TIM_Cmd(LPC_TIM1, ENABLE);
    NVIC_EnableIRQ(TIMER0_IRQn);
    NVIC_EnableIRQ(TIMER1_IRQn);
}

void configUART() {
```

```
    LPC_PINCON->PINSEL0 &= ~(3 << 4); // Clear P0.2 function bits
    LPC_PINCON->PINSEL0 |= (1 << 4); // Set P0.2 as TXD0
    LPC_PINCON->PINSEL0 &= ~(3 << 6); // Clear P0.3 function bits
    LPC_PINCON->PINSEL0 |= (1 << 6); // Set P0.3 as RXD0

    UART_CFG_Type UART;
    UART_ConfigStructInit(&UART);
    UART_Init((LPC_UART_TypeDef *)LPC_UART0, &UART); // Initialize UART0
    UART_IntConfig((LPC_UART_TypeDef *)LPC_UART0, UART_INTCFG_RBR, ENABLE); // Enable RBR interrupt
    NVIC_EnableIRQ(UART0_IRQn);
    UART_TxCmd((LPC_UART_TypeDef *)LPC_UART0, ENABLE); // Enable UART0 Transmit
}

/*
 * INTERRUPTION HANDLERS
 */

void ADC_IRQHandler() {
    // Check which channel triggered the interruption and store its value
    if (LPC_ADC->ADSTAT & (1 << 0)) {
        LDRValue_0 = (LPC_ADC->ADDR0 >> 4) & 0xFFF;
    }
    if (LPC_ADC->ADSTAT & (1 << 1)) {
        LDRValue_1 = (LPC_ADC->ADDR1 >> 4) & 0xFFF;
    }
    if (LPC_ADC->ADSTAT & (1 << 2)) {
        LDRValue_2 = (LPC_ADC->ADDR2 >> 4) & 0xFFF;
    }
    if (LPC_ADC->ADSTAT & (1 << 5)) {
        LDRValue_3 = (LPC_ADC->ADDR5 >> 4) & 0xFFF;
    }
}

void DMA_IRQHandler() {
    GPDMA_ClearIntPending(GPDMA_STATCLR_INTTC, GPDMA_CHANNEL_0); // Clear terminal count interrupt for channel 0
}

void EINT0_IRQHandler() {
    if (debounceFlag == 0) {
        debounceFlag = 1; // Load the debounce counter
        motorEnable_0 =! motorEnable_0;
        motorEnable_1 =! motorEnable_1;
        TIM_Cmd(LPC_TIM0, ENABLE); // Start of TIMER 0
    }
    LPC_SC->EXTINT |= (1 << 0); // Enable the interruption again
}

void EINT1_IRQHandler() {
    if (debounceFlag == 0) {
        debounceFlag = 1; // Load the debounce counter
        errorSelection =! errorSelection; // Toggle error selection for DAC output
        TIM_Cmd(LPC_TIM0, ENABLE); // Start of TIMER 0
    }
    LPC_SC->EXTINT |= (1 << 1); // Enable the interruption again
}

void EINT2_IRQHandler() {
    if (debounceFlag == 0) {
        debounceFlag = 1; // Load the debounce counter
        if (modeSelection == 0) {
            modeSelection = 1; // Switch to joystick mode
            NVIC_DisableIRQ(ADC_IRQn); // Disable ADC interruptions in joystick mode
        } else {
```

```
            modeSelection = 0; // Switch to LDRs mode
            NVIC_EnableIRQ(ADC_IRQn); // Enable ADC interruptions in LDRs mode
        }
        TIM_Cmd(LPC_TIM0, ENABLE); // Start of TIMER 0
    }
    LPC_SC->EXTINT |= (1 << 2); // Enable the interruption again
}

void EINT3_IRQHandler() {
    if (debounceFlag == 0) {
        debounceFlag = 1; // Load the debounce counter

        // - NOT USED -

        TIM_Cmd(LPC_TIM0, ENABLE); // Start of TIMER 0
    }
    LPC_SC->EXTINT |= (1 << 3); // Enable the interruption again
}

void SysTick_Handler() {
    pwmCounter_0++;
    if (pwmCounter_0 >= PWM_CYCLES) {
        pwmCounter_0 = 0; // Reset the PWM counter after a full cycle
    }
    pwmCounter_1++;
    if (pwmCounter_1 >= PWM_CYCLES) {
        pwmCounter_1 = 0; // Reset the PWM counter after a full cycle
    }
    if (joystickCounter > 0) { // Decrease joystick counter if it's active
        joystickCounter--; // Decrease counter
        if (joystickCounter == 0) { // When counter reaches zero, move to next joystick command
            joystickIndex++; // Move to next joystick command
            if (joystickSize > 0 && joystickIndex >= joystickSize) { // End of stored commands reached
                joystickSize = 0; // Clear buffer
                joystickIndex = 0; // Reset index
                for (int i = 0; i < JOYSTICK_BUFFER_SIZE; i++) {
                    joystickBuffer[i] = 0; // Clear buffer contents
                }
            }
        }
    } else { // If joystick counter is not active, check if it have movements to process
        if (joystickIndex >= 0) { // If there is a valid joystick command to process
            joystickCounter = JOYSTICK_CYCLES; // Load joystick counter
        }
    }
}

void TIMER0_IRQHandler() {
    if (TIM_GetIntStatus(LPC_TIM0, TIM_MR0_INT) == 1){
        debounceFlag = 0;
        TIM_ClearIntPending(LPC_TIM0, TIM_MR0_INT);
    }
}

void TIMER1_IRQHandler() {
    if (TIM_GetIntStatus(LPC_TIM1, TIM_MR0_INT) == 1){
        DACUpdateFlag = 1;
        TIM_ClearIntPending(LPC_TIM1, TIM_MR0_INT);
    }
}

void UART0_IRQHandler(void) {
    uint32_t intsrc;
```

```
    uint32_t tmp;
    intsrc = UART_GetIntId((LPC_UART_TypeDef *)LPC_UART0);
    tmp = intsrc & UART_IIR_INTID_MASK;
    if (tmp == UART_IIR_INTID_RDA) { // RDA="Receive Data Available"
        rx_data = UART_ReceiveByte((LPC_UART_TypeDef *)LPC_UART0);
        if (joystickSize < JOYSTICK_BUFFER_SIZE) {
            switch (rx_data) {
                case '0': // No movement
                    UARTSendString((uint8_t *)"0");
                    joystickBuffer[joystickSize] = 0;
                    joystickSize++;
                    break;
                case '1':
                    UARTSendString((uint8_t *)"1");
                    joystickBuffer[joystickSize] = 1;
                    joystickSize++;
                    break;
                case '2':
                    UARTSendString((uint8_t *)"2");
                    joystickBuffer[joystickSize] = 2;
                    joystickSize++;
                    break;
                case '3':
                    UARTSendString((uint8_t *)"3");
                    joystickBuffer[joystickSize] = 3;
                    joystickSize++;
                    break;
                case '4':
                    UARTSendString((uint8_t *)"4");
                    joystickBuffer[joystickSize] = 4;
                    joystickSize++;
                    break;
                case '\r': // End of command
                    UARTSendString((uint8_t *)"r");
                    joystickBuffer[joystickSize] = 0;
                    joystickSize++;
                    break;
                default: // Invalid character received
                    UARTSendString((uint8_t *)"e");
                    joystickBuffer[joystickSize] = 0;
                    joystickSize++;
                    break;
            }
            if (joystickSize == 1) { // If this is the first byte received, reset index to start playback
                joystickIndex = 0;
                joystickCounter = 0;
            }
        } else { // If buffer is full ignore further bytes or optionally wrap
            UARTSendString((uint8_t *)"EOB"); // End Of Buffer
        }
    }
}

/*
 * GENERAL METHODS
 */

void UARTSendString(uint8_t *str) {
    UART_Send((LPC_UART_TypeDef *)LPC_UART0, str, strlen((char *)str), BLOCKING);
}

void processThrottleAndDirection() {
    int diffRightLeft = LDRValue_0 - LDRValue_1;
```

```
    if (diffRightLeft > 0) {
        motorDirection_0 = 0; // Right
    } else {
        motorDirection_0 = 1; // Left
    }
    motorThrottle_0 = calculatePID_0(); // Use PID to determine throttle
    int diffUpDown = LDRValue_2 - LDRValue_3;
    if (diffUpDown > 0) {
        motorDirection_1 = 0; // Up
    } else {
        motorDirection_1 = 1; // Down
    }
    motorThrottle_1 = calculatePID_1(); // Use PID to determine throttle
}

int calculatePID_0() {
    measuredValue_0 = LDRValue_0 - LDRValue_1; // Difference between LDRs
    error_0 = setpoint_0 - measuredValue_0;
    integral_0 += error_0 * (SYSTICK_TIME_IN_US / 1000000.0); // Integrate the error over time
    integral_0 = constrain(integral_0, -WINDUP_LIMIT_0, WINDUP_LIMIT_0); // Prevent integral windup
    derivative_0 = (error_0 - previousError_0) / (SYSTICK_TIME_IN_US / 1000000.0); // Derivative of the error
    output_0 = KP_0 * error_0 + KI_0 * integral_0 + KD_0 * derivative_0;
    previousError_0 = error_0;
    return min(MAX_THROTTLE, max(1, abs((int)output_0))); // Scale throttle based on PID output
}

int calculatePID_1() {
    measuredValue_1 = LDRValue_2 - LDRValue_3; // Difference between LDRs
    error_1 = setpoint_1 - measuredValue_1;
    integral_1 += error_1 * (SYSTICK_TIME_IN_US / 1000000.0); // Integrate the error over time
    integral_1 = constrain(integral_1, -WINDUP_LIMIT_1, WINDUP_LIMIT_1); // Prevent integral windup
    derivative_1 = (error_1 - previousError_1) / (SYSTICK_TIME_IN_US / 1000000.0); // Derivative of the error
    output_1 = KP_1 * error_1 + KI_1 * integral_1 + KD_1 * derivative_1;
    previousError_1 = error_1;
    return min(MAX_THROTTLE, max(1, abs((int)output_1))); // Scale throttle based on PID output
}

void updateMotor0() {
    if (motorDirection_0) {
        LPC_GPIO2->FIOCLR |= (1 << 0);
        LPC_GPIO2->FIOSET |= (1 << 1);
    } else {
        LPC_GPIO2->FIOSET |= (1 << 0);
        LPC_GPIO2->FIOCLR |= (1 << 1);
    }
    if (motorEnable_0) {
        if (pwmCounter_0 < (motorThrottle_0 * (PWM_CYCLES / MAX_THROTTLE))) {
            LPC_GPIO2->FIOSET |= (1 << 4);
        } else {
            LPC_GPIO2->FIOCLR |= (1 << 4);
        }
    } else {
        LPC_GPIO2->FIOCLR |= (1 << 4);
    }
}

void updateMotor1() {
    if (motorDirection_1) {
        LPC_GPIO2->FIOCLR |= (1 << 2);
        LPC_GPIO2->FIOSET |= (1 << 3);
    } else {
        LPC_GPIO2->FIOSET |= (1 << 2);
        LPC_GPIO2->FIOCLR |= (1 << 3);
```

```
    }
    if (motorEnable_1) {
        if (pwmCounter_1 < (motorThrottle_1 * (PWM_CYCLES / MAX_THROTTLE))) {
            LPC_GPIO2->FIOSET |= (1 << 5);
        } else {
            LPC_GPIO2->FIOCLR |= (1 << 5);
        }
    } else {
        LPC_GPIO2->FIOCLR |= (1 << 5);
    }
}

void updateDAC() {
    if (errorSelection == 0) {
        DACValue = constrain((int)(error_0 + 512), 0, 1023); // Output error_0 centered at 512
    } else {
        DACValue = constrain((int)(error_1 + 512), 0, 1023); // Output error_1 centered at 512
    }
    GPDMA_ChannelCmd(GPDMA_CHANNEL_0, ENABLE); // Start GPDMA transfer to update DAC
}
```