#include <LiquidCrystal.h> // Librería para manejar el LCD

// Configuración de pines del LCD (12, 11, 5, 4, 3, 2)

LiquidCrystal pantalla(12, 11, 5, 4, 3, 2);

// Definición de pines y variables globales

const int pinAnalog = A0; // Entrada para la señal analógica

const int pinStart = 7; // Botón de inicio

const int pinDisplay = 8; // Botón para mostrar información

// Variables para la captura de la señal

float\* bufferAmplitudes; // Puntero para arreglo de amplitudes dinámicas

float\* bufferDerivadas; // Puntero para el arreglo de segundas derivadas

int numMuestras = 60; // Número de muestras

float valorMax = 0;

float valorMin = 1023;

float valorAmplitud = 0;

float valorFrecuencia = 0;

unsigned long tiempoPrevio = 0;

String tipoOnda = "No identificada"; // Variable para almacenar la forma de la onda

// Control para mostrar en pantalla

bool alternarPantalla = true; // Alternar entre diferentes características

// Inicialización del sistema

void inicializarSistema() {

// Configuración del LCD

pantalla.begin(16, 2);

pantalla.print("Iniciando...");

// Configuración de los pines de los botones

pinMode(pinStart, INPUT\_PULLUP);

pinMode(pinDisplay, INPUT\_PULLUP);

// Inicialización de memoria dinámica

bufferAmplitudes = new float[numMuestras];

bufferDerivadas = new float[numMuestras - 2];

// Comunicación serial

Serial.begin(9600);

}

// Bucle principal

void loop() {

if (digitalRead(pinStart) == LOW) {

pantalla.clear();

pantalla.print("Capturando...");

capturarDatos(); // Captura las 60 muestras cuando el botón es presionado

delay(200);

}

if (digitalRead(pinDisplay) == LOW) {

desplegarDatos(); // Muestra resultados en pantalla

delay(200);

}

}

// Función para capturar la señal analógica

void capturarDatos() {

for (int i = 0; i < numMuestras; i++) {

bufferAmplitudes[i] = analogRead(pinAnalog);

Serial.println(bufferAmplitudes[i]);

// Actualiza los valores máximos y mínimos

if (bufferAmplitudes[i] > valorMax) {

valorMax = bufferAmplitudes[i];

}

if (bufferAmplitudes[i] < valorMin) {

valorMin = bufferAmplitudes[i];

}

// Cálculo del tiempo entre picos para la frecuencia

unsigned long tiempoActual = millis();

if (bufferAmplitudes[i] > 512 && tiempoActual - tiempoPrevio > 20) {

valorFrecuencia = 1000.0 / (tiempoActual - tiempoPrevio);

tiempoPrevio = tiempoActual;

}

delay(10);

}

}

// Función para identificar la forma de la onda usando la segunda derivada

void identificarOnda() {

int contadorDerivadaMenor5 = 0; // Contador para derivadas menores a 5

for (int i = 2; i < numMuestras; i++) {

float segundaDerivada = abs(bufferAmplitudes[i]) - 2 \* abs(bufferAmplitudes[i - 1]) + abs(bufferAmplitudes[i - 2]);

bufferDerivadas[i - 2] = abs(segundaDerivada);

if (abs(segundaDerivada) < 5) {

contadorDerivadaMenor5++;

}

}

Serial.println("Derivadas:");

for (int i = 0; i < numMuestras - 2; i++) {

Serial.print(bufferDerivadas[i]);

if (i < numMuestras - 3) Serial.print(", ");

}

Serial.println();

// Clasificación de la forma de la onda

if (contadorDerivadaMenor5 > 10) {

tipoOnda = "Triangular";

} else {

tipoOnda = "Senoidal";

}

}

// Función para mostrar las características en la pantalla

void mostrarPantalla() {

if (alternarPantalla) {

pantalla.clear();

pantalla.setCursor(0, 0);

pantalla.print("Onda: ");

pantalla.setCursor(0, 1);

pantalla.print(tipoOnda);

} else {

pantalla.clear();

pantalla.setCursor(0, 0);

pantalla.print("Amp: ");

pantalla.print(valorAmplitud);

pantalla.print("V");

pantalla.setCursor(0, 1);

pantalla.print("Freq: ");

pantalla.print(valorFrecuencia);

pantalla.print(" Hz");

}

alternarPantalla = !alternarPantalla;

delay(2000);

}

// Función para desplegar los resultados

void desplegarDatos() {

identificarOnda();

valorAmplitud = (valorMax - valorMin) \* (5.0 / 1023.0);

int ciclos = 0;

while (digitalRead(pinStart) == HIGH && ciclos != 6) {

ciclos++;

mostrarPantalla();

}

pantalla.clear();

pantalla.setCursor(0, 0);

pantalla.print("Presiona: INICIO");

// Reiniciar variables

valorMax = 0;

valorMin = 1023;

tiempoPrevio = 0;

}

// Función para liberar memoria dinámica

void liberarRecursos() {

delete[] bufferAmplitudes;

delete[] bufferDerivadas;

}