// Код программы

// BluetoothCar.ino

#define MOTOR\_MAX 255

#define JOY\_MAX 40

#define minDuty 0

#define RIGHT\_MOTOR\_DIRECTION NORMAL

#define LEFT\_MOTOR\_DIRECTION NORMAL

#define RIGHT\_MOTOR\_MODE HIGH

#define LEFT\_MOTOR\_MODE HIGH

#define RIGHT\_MOTOR\_D 10

#define RIGHT\_MOTOR\_PWM 11

#define LEFT\_MOTOR\_D 2

#define LEFT\_MOTOR\_PWM 3

#define BT\_TX 13

#define BT\_RX 12

#include "Motor.h"

GMotor motorR(DRIVER2WIRE, RIGHT\_MOTOR\_D, RIGHT\_MOTOR\_PWM, RIGHT\_MOTOR\_MODE);

GMotor motorL(DRIVER2WIRE, LEFT\_MOTOR\_D, LEFT\_MOTOR\_PWM, LEFT\_MOTOR\_MODE);

#include <SoftwareSerial.h>

SoftwareSerial BTserial(BT\_TX, BT\_RX);

boolean doneParsing, startParsing;

int dataX, dataY;

String string\_convert;

void setup() {

#if (LEFT\_MOTOR\_PWM == 3 || LEFT\_MOTOR\_PWM == 11 || RIGHT\_MOTOR\_PWM == 3 || RIGHT\_MOTOR\_PWM == 11)

TCCR2B = 0b00000001;

TCCR2A = 0b00000011;

#elif (LEFT\_MOTOR\_PWM == 9 || LEFT\_MOTOR\_PWM == 10 || RIGHT\_MOTOR\_PWM == 9 || RIGHT\_MOTOR\_PWM == 10)

TCCR1A = 0b00000001;

TCCR1B = 0b00001001;

#endif

Serial.begin(9600);

BTserial.begin(9600);

motorR.setMode(AUTO);

motorL.setMode(AUTO);

motorR.setMinDuty(minDuty);

motorL.setMinDuty(minDuty);

motorR.setDirection(RIGHT\_MOTOR\_DIRECTION);

motorL.setDirection(LEFT\_MOTOR\_DIRECTION);

}

void loop() {

parsing();

if (doneParsing) {

doneParsing = false;

int joystickX = map((dataX), -JOY\_MAX, JOY\_MAX, -MOTOR\_MAX / 2, MOTOR\_MAX / 2);

int joystickY = map((dataY), -JOY\_MAX, JOY\_MAX, -MOTOR\_MAX, MOTOR\_MAX);

int dutyR = joystickY + joystickX;

int dutyL = joystickY - joystickX;

dutyR = constrain(dutyR, -MOTOR\_MAX, MOTOR\_MAX);

dutyL = constrain(dutyL, -MOTOR\_MAX, MOTOR\_MAX);

motorR.smoothTick(dutyR);

motorL.smoothTick(dutyL);

}

}

void parsing() {

if (BTserial.available() > 0) {

char incomingChar = BTserial.read();

Serial.println(incomingChar);

if (startParsing) {

if (incomingChar == ' ') {

dataX = string\_convert.toInt();

string\_convert = "";

Serial.print("DataX = ");

Serial.println(dataX);

}

else if (incomingChar == ';') {

dataY = string\_convert.toInt();

string\_convert = "";

startParsing = false;

doneParsing = true;

Serial.print("DataY = ");

Serial.println(dataY);

} else {

string\_convert += incomingChar;

}

}

if (incomingChar == '$') {

startParsing = true;

}

}

}

// Motor.h

#ifndef Motor\_h

#define Motor\_h

#include <Arduino.h>

#define \_SMOOTH\_PRD 50

enum M\_driverType {

DRIVER2WIRE\_NO\_INVERT,

DRIVER2WIRE,

DRIVER3WIRE,

RELAY2WIRE,

};

#define NORMAL 0

#define REVERSE 1

enum M\_workMode {

FORWARD,

BACKWARD,

STOP,

BRAKE,

AUTO = 0,

};

static const int8\_t \_M\_NC = -1; // not connected

class Motor {

public:

Motor(M\_driverType type, int8\_t param1 = \_M\_NC, int8\_t param2 = \_M\_NC, int8\_t param3 = \_M\_NC, int8\_t param4 = \_M\_NC);

void setSpeed(int16\_t duty);

void setMode(M\_workMode mode);

void setDirection(bool direction);

void setMinDuty(int duty);

void setResolution(byte bit);

void setDeadtime(uint16\_t deadtime);

void setLevel(int8\_t level);

void smoothTick(int16\_t duty);

void setSmoothSpeed(uint8\_t speed);

int getState();

int16\_t \_duty = 0;

void set8bitMode();

void set10bitMode();

protected:

void setPins(bool a, bool b, int c);

void run(M\_workMode mode, int16\_t duty = 0);

int16\_t \_dutyS = 0;

int \_minDuty = 0, \_state = 0;;

int8\_t \_digA = \_M\_NC, \_digB = \_M\_NC, \_pwmC = \_M\_NC;

bool \_direction = false;

int8\_t \_level = LOW;

int \_maxDuty = 255;

M\_workMode \_mode = FORWARD, \_lastMode = FORWARD;

M\_driverType \_type;

uint16\_t \_deadtime = 0;

uint8\_t \_speed = 20;

uint32\_t \_tmr = 0;

float \_k;

};

#endif

// Motor.cpp

#include "Motor.h"

Motor::Motor(M\_driverType type, int8\_t param1, int8\_t param2, int8\_t param3, int8\_t param4) {

\_type = type;

switch (\_type) {

case DRIVER2WIRE\_NO\_INVERT:

case DRIVER2WIRE:

\_digA = param1;

\_pwmC = param2;

if (param3 != \_M\_NC) \_level = !param3;

break;

case DRIVER3WIRE:

\_digA = param1;

\_digB = param2;

\_pwmC = param3;

if (param4 != \_M\_NC) \_level = !param4;

break;

case RELAY2WIRE:

\_digA = param1;

\_digB = param2;

if (param3 != \_M\_NC) \_level = !param3;

break;

}

if (\_digA != \_M\_NC) pinMode(\_digA, OUTPUT);

if (\_digB != \_M\_NC) pinMode(\_digB, OUTPUT);

if (\_pwmC != \_M\_NC) pinMode(\_pwmC, OUTPUT);

setMode(STOP);

}

void Motor::setSpeed(int16\_t duty) {

if (\_mode < 2) {

\_duty = constrain(duty, -\_maxDuty, \_maxDuty);

if (\_maxDuty > 255 && abs(\_duty) == 255) \_duty++;

if (duty == 0) run(STOP, 0);

else {

if (duty > 0) {

if (\_minDuty != 0) \_duty = \_duty \* \_k + \_minDuty;

run(\_mode, \_duty);

} else {

if (\_minDuty != 0) \_duty = \_duty \* \_k - \_minDuty;

run(BACKWARD, -\_duty);

}

}

}

}

void Motor::run(M\_workMode mode, int16\_t duty) {

// дедтайм

if (\_deadtime > 0 && \_lastMode != mode) {

\_lastMode = mode;

setPins(\_level, \_level, 0);

delayMicroseconds(\_deadtime);

}

if (\_direction) {

if (mode == FORWARD) mode = BACKWARD;

else if (mode == BACKWARD) mode = FORWARD;

}

switch (mode) {

case FORWARD: setPins(\_level, !\_level, duty); \_state = 1; break;

case BACKWARD: setPins(!\_level, \_level, (\_type == DRIVER2WIRE) ? (\_maxDuty - duty) : (duty)); \_state = -1; break;

case BRAKE: setPins(!\_level, !\_level, !\_level \* 255); \_state = 0; break;

case STOP: setPins(\_level, \_level, \_level \* 255); \_state = 0; break;

}

}

void Motor::setPins(bool a, bool b, int c) {

if (\_digA != \_M\_NC) digitalWrite(\_digA, a);

if (\_digB != \_M\_NC) digitalWrite(\_digB, b);

if (\_pwmC != \_M\_NC) analogWrite(\_pwmC, c);

}

void Motor::smoothTick(int16\_t duty) {

if (millis() - \_tmr >= \_SMOOTH\_PRD) {

\_tmr = millis();

if (abs(\_dutyS - duty) > \_speed) \_dutyS += (\_dutyS < duty) ? \_speed : -\_speed;

else \_dutyS = duty;

setSpeed(\_dutyS);

}

}

int Motor::getState() {

return \_state;

}

void Motor::setResolution(byte bit) {

\_maxDuty = (1 << bit) - 1;

setMinDuty(\_minDuty);

}

void Motor::setMinDuty(int duty) {

\_minDuty = duty;

\_k = 1.0 - (float)\_minDuty / \_maxDuty;

}

void Motor::setMode(M\_workMode mode) {

\_mode = mode;

run(mode, \_duty);

}

void Motor::setSmoothSpeed(uint8\_t speed) {

\_speed = speed;

}

void Motor::setDirection(bool direction) {

\_direction = direction;

}

void Motor::setDeadtime(uint16\_t deadtime) {

\_deadtime = deadtime;

}

void Motor::setLevel(int8\_t level) {

\_level = !level;

}

// совместимость

void Motor::set8bitMode() {

setResolution(8);

}

void Motor::set10bitMode() {

setResolution(10);

}

Мобильное приложение

MainActivity.kt

class MainActivity : AppCompatActivity() {

private lateinit var navController: NavController

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.fragment\_main)

val navHostFragment = supportFragmentManager

.findFragmentById(R.id.fragmentContainerView) as NavHostFragment

navController = navHostFragment.navController

}

}

JoystickView.kt

@SuppressLint("ClickableViewAccessibility")

class JoystickView @JvmOverloads constructor(context: Context, attrs: AttributeSet? = null, defStyleAttr: Int = 0) : View(context, attrs, defStyleAttr) {

// Constants

private val DEFAULT\_RADIUS = 300f

private val DEFAULT\_COLOR = Color.BLACK

private val DEFAULT\_THUMB\_RADIUS = 130f

private val DEFAULT\_THUMB\_COLOR = Color.RED

// Paint objects

private val circlePaint = Paint().apply {

color = DEFAULT\_COLOR

style = Paint.Style.FILL

}

private val thumbPaint = Paint().apply {

color = DEFAULT\_THUMB\_COLOR

style = Paint.Style.FILL

}

// Circle properties

private var radius = DEFAULT\_RADIUS

private val center = PointF()

// Thumb properties

private var thumbRadius = DEFAULT\_THUMB\_RADIUS

private val thumbPosition = PointF()

// Listener

private var listener: ((angle: Float, strength: Float) -> Unit)? = null

init {

// Set up touch listener

setOnTouchListener { \_, event ->

// Calculate distance and angle from center

val distance = sqrt((event.x - center.x).pow(2) + (event.y - center.y).pow(2))

val angle = (atan2(event.y - center.y, event.x - center.x) \* 180 / PI).toFloat()

when (event.action) {

MotionEvent.ACTION\_DOWN, MotionEvent.ACTION\_MOVE -> {

// Clamp distance to maximum radius

val clampedDistance = min(distance, radius)

// Calculate thumb position based on angle and clamped distance

thumbPosition.x = (center.x + clampedDistance \* cos(angle \* PI / 180)).toFloat()

thumbPosition.y = (center.y + clampedDistance \* sin(angle \* PI / 180)).toFloat()

// Call listener with angle and normalized strength

listener?.invoke(angle, clampedDistance / radius)

invalidate()

}

MotionEvent.ACTION\_UP -> {

// Reset thumb position

thumbPosition.x = center.x

thumbPosition.y = center.y

// Call listener with zero angle and zero strength

listener?.invoke(0f, 0f)

invalidate()

}

}

true

}

}

override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {

// Calculate desired size based on radius

val desiredWidth = (radius \* 2).toInt() + paddingLeft + paddingRight

val desiredHeight = (radius \* 2).toInt() + paddingTop + paddingBottom

// Calculate actual size based on spec

val width = resolveSize(desiredWidth, widthMeasureSpec)

val height = resolveSize(desiredHeight, heightMeasureSpec)

// Take the smaller of the two sizes as the final size

val size = min(width, height)

// Set actual size and center point

setMeasuredDimension(size, size)

center.set(size / 2f, size / 2f)

}

override fun onDraw(canvas: Canvas) {

// Draw circle

canvas.drawCircle(center.x, center.y, radius, circlePaint)

// Draw thumb

canvas.drawCircle(thumbPosition.x, thumbPosition.y, thumbRadius, thumbPaint)

}

fun setOnMoveListener(listener: (angle: Float, strength: Float) -> Unit) {

this.listener = listener

}

fun setThumbRadius(radius: Float) {

this.thumbRadius = radius

invalidate()

}

fun setCircleRadius(radius: Float) {

this.radius = radius

requestLayout()

invalidate()

}

fun setThumbColor(color: Int) {

thumbPaint.color = color

invalidate()

}

fun setCircleColor(color: Int) {

circlePaint.color = color

invalidate()

}

}

JoystickFragment.kt

class JoystickFragment : Fragment() {

private lateinit var joystickView: JoystickView

private lateinit var coordinatesTextView: TextView

private lateinit var btStatusTextView: TextView

private lateinit var bluetoothChecker: BluetoothChecker

@SuppressLint("MissingInflatedId", "SetTextI18n")

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

val view = inflater.inflate(R.layout.fragment\_joystick, container, false)

val button = view.findViewById<Button>(R.id.button)

joystickView = view.findViewById(R.id.joystick\_view)

coordinatesTextView = view.findViewById(R.id.coordinates\_text\_view)

btStatusTextView = view.findViewById(R.id.bt\_status\_text\_view)

bluetoothChecker = BluetoothChecker()

bluetoothChecker.start()

button.setOnClickListener{

findNavController().navigate(R.id.action\_joystickFragment\_to\_settingsFragment)

}

joystickView.setOnMoveListener { angle, strength ->

val x = (41 \* cos(angle \* PI / 180) \* strength).toInt()

val y = (41 \* -sin(angle \* PI / 180) \* strength).toInt()

coordinatesTextView.text = "($x, $y)"

if (BluetoothConstant.connectedDevice != null) {

CoroutineScope(Dispatchers.IO).launch {

sendCoordinates(x, y)

delay(50)

}

btStatusTextView.text = "Connected"

}

else

btStatusTextView.text = "Not Connected"

}

return view

}

override fun onDestroy() {

super.onDestroy()

bluetoothChecker.stop()

}

private fun sendCoordinates(x: Int, y: Int) {

try {

val message = "$x,$y"

val correctMessage = "$$x $y;"

BluetoothConstant.outputStream?.write(correctMessage.toByteArray())

Log.i(TAG, "Sending: ${message.toByteArray()})")

Log.i(TAG, "RAW message: $correctMessage")

} catch (e: IOException) {

Log.e(TAG, "Failed to send coordinates to Bluetooth device", e)

}

}

companion object {

private const val TAG = "JoystickFragment"

}

}

BluetoothChecker.kt

class BluetoothChecker {

private var isChecking = false

private val checkIntervalMillis = 5000L // 5 seconds

private var job: Job? = null

fun start() {

isChecking = true

job = GlobalScope.launch {

while (isChecking) {

if (!isBluetoothConnected()) {

closeConnections()

return@launch

}

delay(checkIntervalMillis)

}

}

}

fun stop() {

isChecking = false

job?.cancel()

}

private suspend fun isBluetoothConnected(): Boolean {

return try {

withContext(Dispatchers.IO) {

BluetoothConstant.outputStream?.write(0)

}

true

} catch (e: Exception) {

false

}

}

private suspend fun closeConnections() {

withContext(Dispatchers.IO) {

try {

BluetoothConstant.outputStream?.close()

BluetoothConstant.inputStream?.close()

BluetoothConstant.bluetoothSocket?.close()

BluetoothConstant.connectedDevice = null

} catch (e: Exception) {

Log.e(TAG, "$e")

}

}

}

companion object {

private const val TAG = "BluetoothChecker"

}

}

SettingsFragment.kt

class SettingsFragment : Fragment() {

private lateinit var bluetoothAdapter: BluetoothAdapter

@SuppressLint("MissingInflatedId")

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

val view = inflater.inflate(R.layout.fragment\_settings, container, false)

// Get a reference to the BluetoothAdapter

bluetoothAdapter = BluetoothAdapter.getDefaultAdapter()

// Check if Bluetooth is supported and enabled

if (!bluetoothAdapter.isEnabled) {

// Bluetooth is not supported or not enabled, do nothing

return view

}

// Query the paired devices

if (ActivityCompat.checkSelfPermission(

requireContext(),

Manifest.permission.BLUETOOTH\_CONNECT

) != PackageManager.PERMISSION\_GRANTED

) {

return view

}

val pairedDevices = bluetoothAdapter.bondedDevices

// Create a list of device names and MAC addresses

val deviceList = ArrayList<String>()

for (device in pairedDevices) {

deviceList.add(device.name + "\n" + device.address)

}

// Display the list in a ListView

val listView = view.findViewById<ListView>(R.id.device\_list\_view)

val adapter = ArrayAdapter(requireContext(), android.R.layout.simple\_list\_item\_1, deviceList)

listView.adapter = adapter

// Set a click listener for the list items

listView.setOnItemClickListener { parent, view, position, id ->

val deviceInfo = deviceList[position].split("\n")

val deviceName = deviceInfo[0]

val deviceAddress = deviceInfo[1]

try {

connectToBluetoothDevice(deviceAddress)

Toast.makeText(requireContext(), "Selected device: $deviceName ($deviceAddress)", Toast.LENGTH\_SHORT).show()

}

catch (exception: Exception) {

Toast.makeText(requireContext(), "Can't connected to device", Toast.LENGTH\_SHORT).show()

}

}

return view

}

private fun connectToBluetoothDevice(deviceAddress: String) {

bluetoothAdapter = BluetoothAdapter.getDefaultAdapter()

val hc06Device = bluetoothAdapter.getRemoteDevice(deviceAddress)

val uuid = UUID.fromString("00001101-0000-1000-8000-00805F9B34FB")

if (ActivityCompat.checkSelfPermission(

requireContext(),

Manifest.permission. BLUETOOTH\_CONNECT

) != PackageManager.PERMISSION\_GRANTED

) {

Log.e(TAG, "No permissions to connect BT")

return

}

val bluetoothSocket = hc06Device.createInsecureRfcommSocketToServiceRecord(uuid)

bluetoothSocket.connect()

BluetoothConstant.outputStream = bluetoothSocket.outputStream

BluetoothConstant.inputStream = bluetoothSocket.inputStream

BluetoothConstant.connectedDevice = hc06Device

BluetoothConstant.bluetoothSocket = bluetoothSocket

}

companion object {

private const val TAG = "SettingsFragment"

}

}

Bluetooth.kt

object BluetoothConstant {

var outputStream: OutputStream? = null

var inputStream: InputStream? = null

var connectedDevice: BluetoothDevice? = null

var bluetoothSocket: BluetoothSocket? = null

}