01

LINE FOLLOWER

15 min

https://github.com/andreibexa/Line-Follower-Arduino

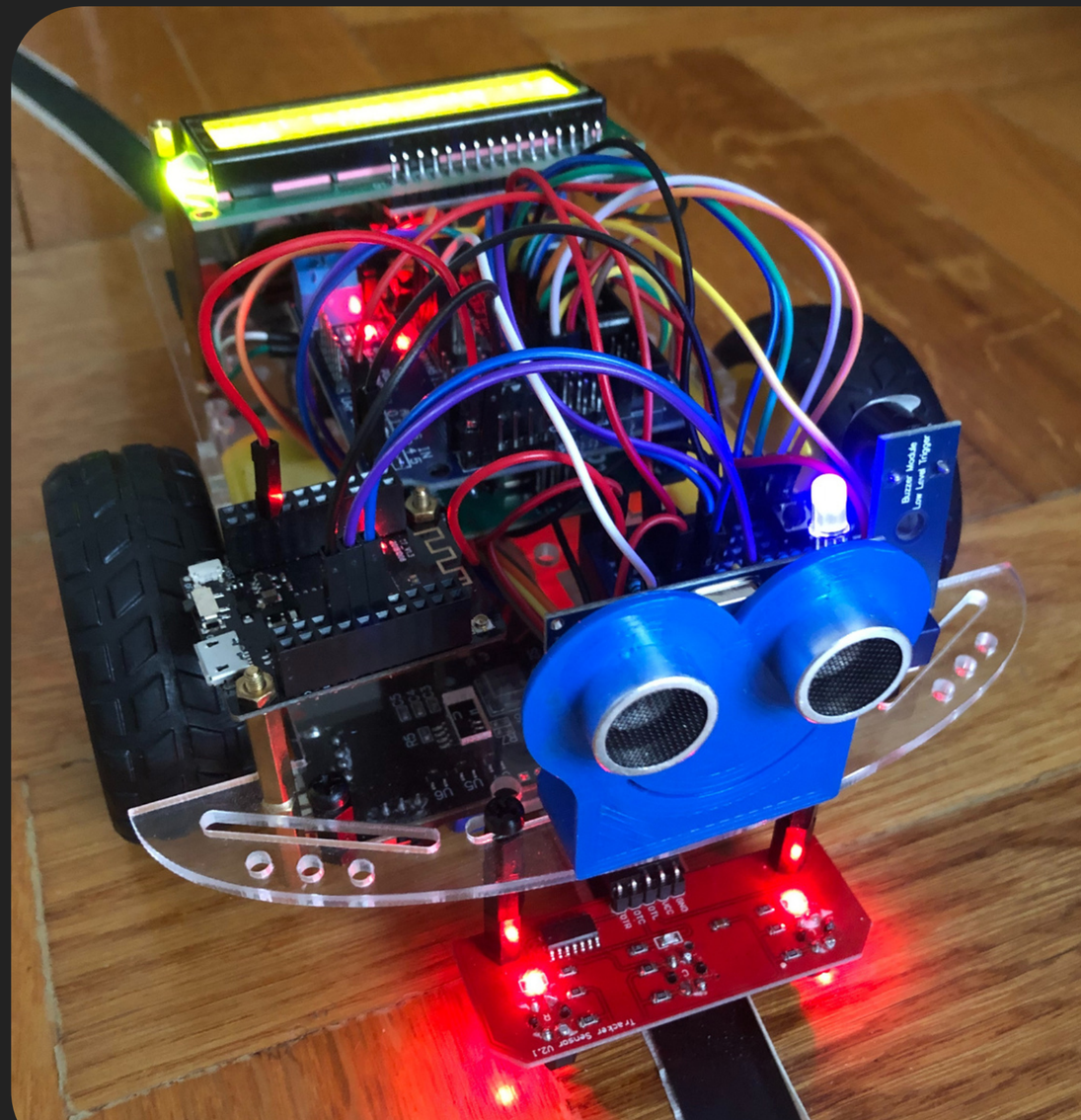


Table of contents

02

Minimum requirements

1. Line Follower
2. Obstacle avoider
3. Documentation

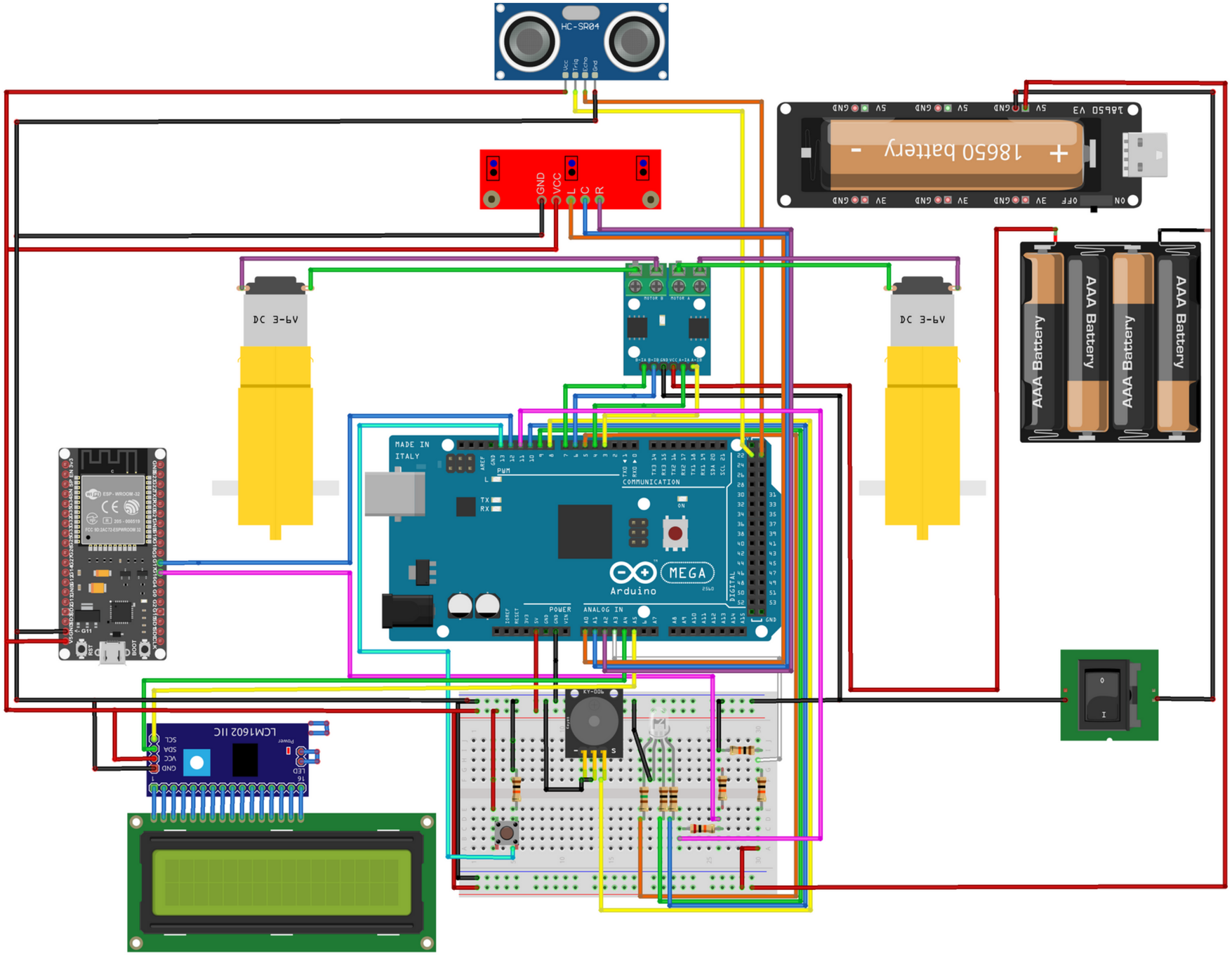
04

Main functionalities

1. Line Follower
2. Line obstacle avoider
3. Communication between 2 microcontrollers
4. WiFi Manager
5. Configuration and control through Arduino Cloud
6. Configuration saving in EEPROM
7. Push button - Line Follower start/stop
8. LCD Display:
    - RSSI - WiFi signal strength
    - Cloud status
    - 18650 battery voltage
    - AA batteries voltage
9. RGB LED for: WiFi, Cloud, and Line Follower status
10. Buzzer
11. System power - on/off button

Diagram

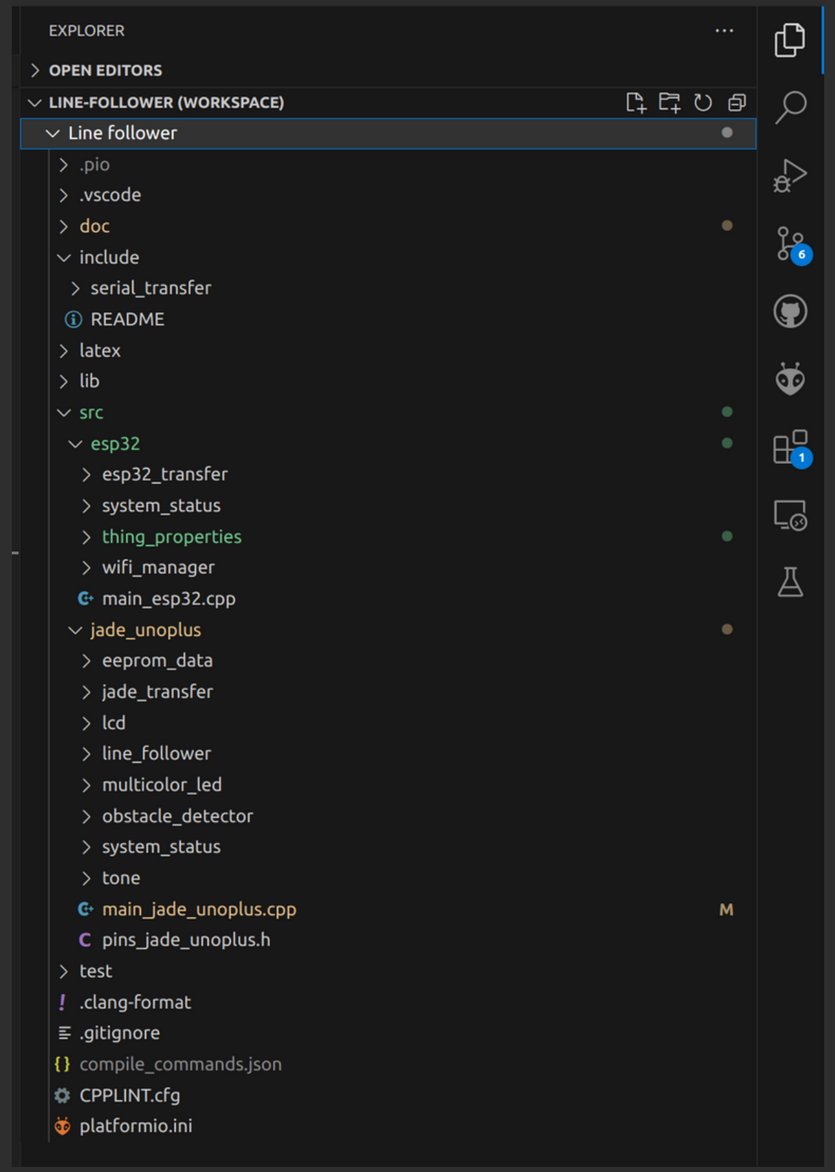
05


HC-SR04
GND VCC L C R
18650 battery
DC 3-6V
DC 3-6V
AAA Battery
AAA Battery
AAA Battery
AAA Battery
MADE IN ITALY
PWM
COMMUNICATION
MEGA
Arduino
POWER
ANALOG IN
DIGITAL
ESP-WROOM-32
LCM1602 IIC
GND VCC SDA SCL
LED
KY-004

Component list

06

1. 2WD Car Chassis Kit, Line Follower
2. Groundstudio Jade Uno+ (ATmega328PB)
3. LilyGO TTGO T7 Mini32 V1.3 (ESP32)
4. Sensor Shield v5.0 for Arduino
5. 3-channel Infrared Line Tracking Module (CTRT5000)
6. Ultrasonic Sensor Module - HC-SR04 distance detector
7. 1602 Green LCD with I2C Interface
8. H-Bridge Motor Driver - L9110S
9. DC Geared Motor 1:48, 3V-6V
10. 18650 Battery, 3.7V, 2200mAh
11. 1-Cell 18650 Battery Charging Module, 5V Step-up Converter
12. Set of 4 AA Batteries, 1.2V, 2200mAh
13. RGB LED
14. Passive Buzzer
15. Tactile Push Button
16. Simple On/Off Switch
17. Resistors 10kΩ, 20kΩ, 100kΩ, and 150kΩ
18. Dupont Female-Female Jumper Wires
19. Dupont Female-Male Jumper Wires
20. Standoffs, Screws, Nuts - M3
21. Standoffs - M2.5



VSCode + PlatformIO

Libraries for Jade Uno+

Wire
powerbroker2/SerialTransfer
marcoschwartz/LiquidCrystal_I2C

Libraries for ESP32

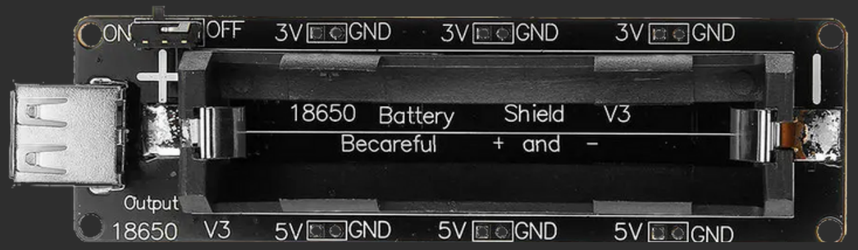
FS
Wire
WiFi
Update
WebServer
DNSServer
powerbroker2/SerialTransfer
arduino-libraries/ArduinoIoTCloud
arduino-libraries/Arduino_ESP32_OTA
https://github.com/tzapu/WiFiManager.git

08

Power sources

18650 Battery Charging Module

The components, except for the H-Bridge, are powered through the charging module of a LI-ION battery, type 18650, 3.7V, 2200mAh. The module has an output of 5V - 2A and 3V - 1A.



4 rechargeable AA batteries, 1.2V

For powering the H-Bridge and the two DC motors, 4 Ni-MH rechargeable batteries, type AA, 1.2V, 2500mAh are used.

09

Line follower

Presentation

A line tracking module with 3 digital IR sensors was used. The distance between sensors is 20mm.

For calculating the motor speeds, a simplified PID control algorithm was employed, utilizing only the proportional coefficient (Kp), without the derivative coefficient (Kd) and integral coefficient (Ki).

Starting the line lollower

Push button

The Line Follower can be started by pressing the push button on the breadboard, which toggles the lineFollowerMode variable.

Additionally, it can also be started from the Arduino Cloud.

```
1
2  // Current button state
3  int buttonState;
4  // Previous button state
5  int lastButtonState;
6  // Time of last button state change
7  unsigned long lastDebounceTime = 0;
8  // Debounce time in milliseconds
9  const unsigned long debounceDelay = 50;
10
11 /**
12  * @brief Debounce the line follower mode on button pushup
13  *
14  */
15 void debounceLineFollowerButton() {
16   // Read the current button state
17   int reading = digitalRead(LINE_FOLLOWER_BUTTON_PIN);
18
19   // Check for button state change with debounce
20   if (reading != lastButtonState) {
21     // Update the time of last button state change
22     lastDebounceTime = millis();
23   }
24
25   // Check if enough debounce time has passed
26   if ((millis() - lastDebounceTime) > debounceDelay) {
27     // Check if the button state has actually changed
28     if (reading != buttonState) {
29       buttonState = reading;
30
31       // Only toggle the LED if the new button state is HIGH
32       if (buttonState == HIGH) {
33         // Button was pressed
34         lineFollowerMode = !lineFollowerMode;
35       }
36     }
37   }
38
39   // Update the previous button state
40   lastButtonState = reading;
41 }
```

Runnig
the line follower

The following instructions are executed in a loop:

- The buzzer emits a sound, and the LED changes its color to green.
- The sensor values are read, and the position of the line is calculated.
- If the obstacle avoidance mode is activated, the robot avoids obstacles.
- The sensor values are read again, and the line's position is calculated after avoiding the obstacle.
- The motor speeds are calculated and adjusted using the PID algorithm.
- The robot checks if the finish line is detected.

```
1  void runLineFollower() {
2    if (isRunning == false) {
3      // Change the led color to Green
4      setMultiColorLed(0, 10, 0, false);
5
6      // Play a sound when the line follower starts
7      playSequence(S_BUTTON_PUSHED);
8    }
9
10   isRunning = true;
11
12   // Read line position
13   readLinePosition();
14
15   // Avoid obstacle closer than 30 cm
16   if (avoidObstacleMode == true) {
17     avoidObstacle(30);
18   }
19
20   // Read line position after obstacle
21   readLinePosition();
22
23   // Calculate the error position and follow the line
24   calculatePID();
25
26   // Check if the finish line is detected
27   checkFinishLine();
28 }
```

Stopping 🚨 the line follower

Status check

"stopLineFollower()" function is called only once upon receiving the stop command.

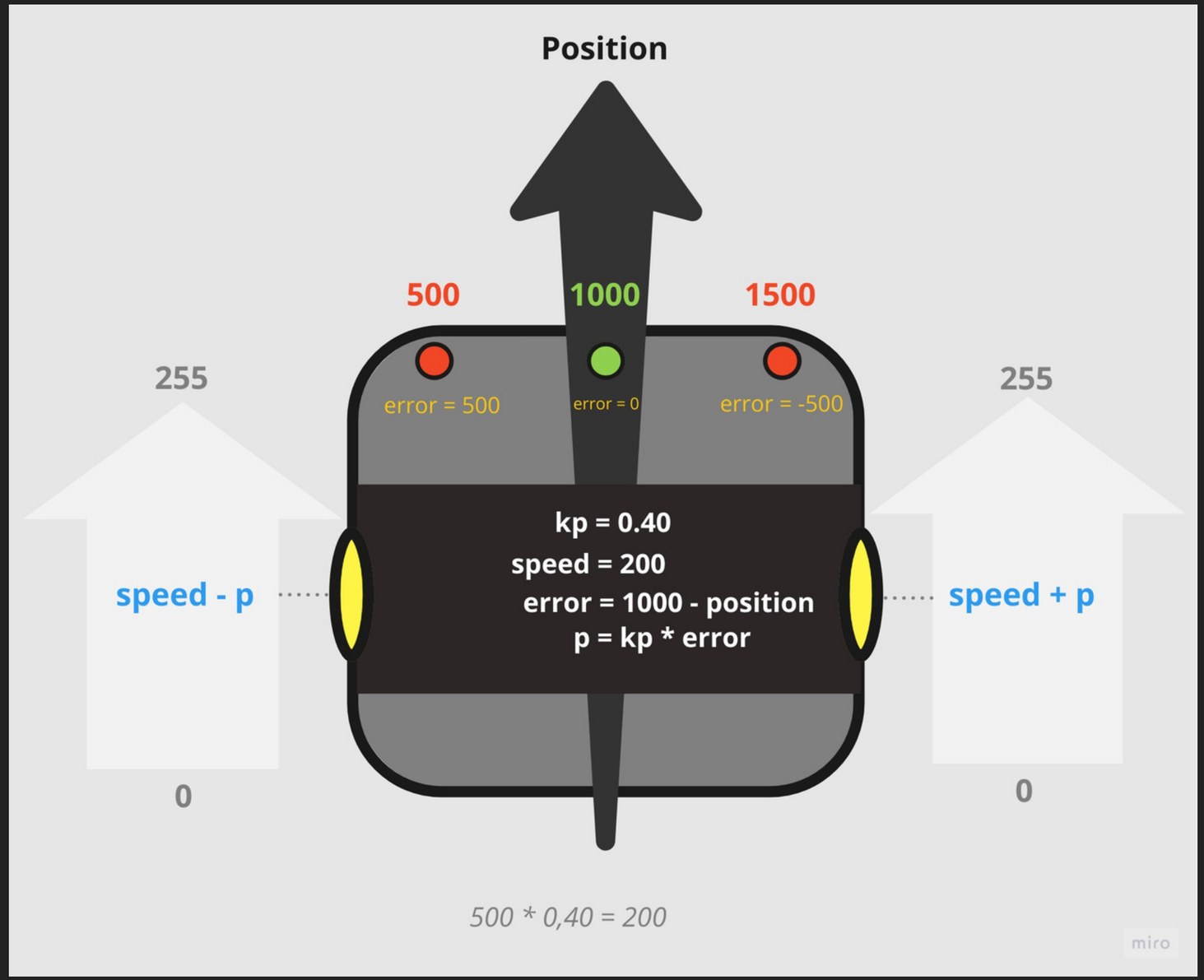
Upon stopping, the following actions are taken:

- Motors are turned off to stop the movement of the robot.
- The LED changes its color to red to indicate that the line follower has stopped.
- The line follower settings are transmitted to the cloud to update the state of the startup switch in the control panel.
- The buzzer emits a sound to provide an audible indication that the line follower has stopped.

```
/**
 * @brief Run the Line Follower only if lineFollowerMode is true
 */
void loopLineFollower() {
  if (lineFollowerMode == true) {
    runLineFollower();
  } else {
    stopLineFollower();
  }
}
```

```
/**
 * @brief Disable the Line Follower
 *
 */
void stopLineFollower() {
  // Avoid running this function in a loop
  if (isRunning == false) {
    return;
  }

  isRunning = false;

  // Stop both motors
  setDirection(STOP_SLOW, 0);

  // Change the led color to Red
  setMultiColorLed(10, 0, 0);

  // Send the lineFollowerMode status to Arduino Cloud
  lineFollowerMode = false;
  transmitLineFollowerSettings();

  playSequence(S_SLEEPING);
}
```

PID controller



Positions

Possible positions: 500, 1000, 1500;

Each sensor returns a digital value:

- 0 for line detected;
- 1 for line not detected.


```
numDetectedLineSensor = lineSensorValues[0] +
                        lineSensorValues[1] +
                        lineSensorValues[2];

currentPosition =
                (
                 2000 * lineSensorValues[2] +
                 1000 * lineSensorValues[1] +
                 0 * lineSensorValues[0]
                ) / numDetectedLineSensor;
```

Simplified PID control

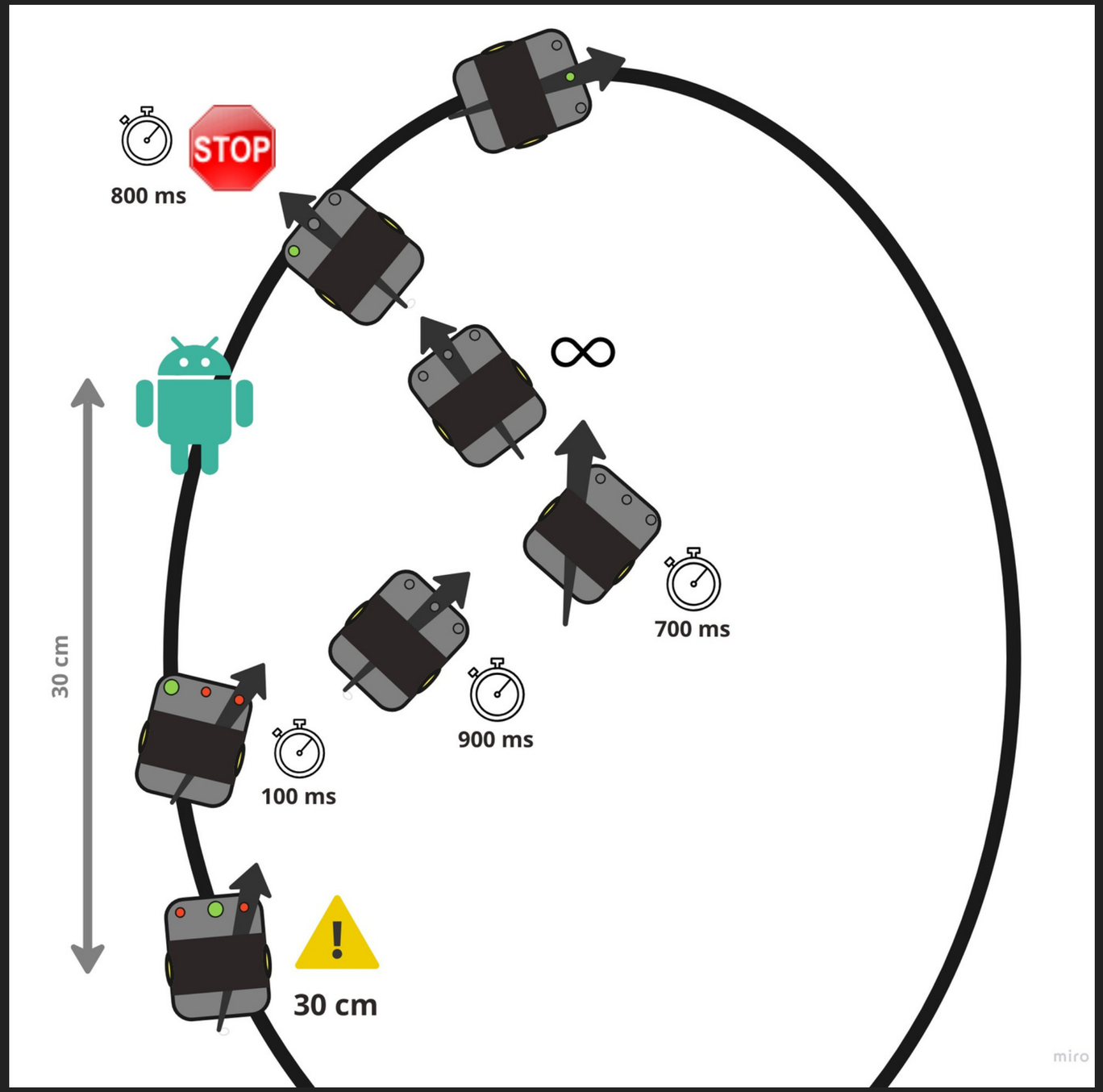
- Kp adjusts the responsiveness and oscillation of the system.
- Motor speed is limited between 0 and 255.
- Motor speed is proportional to the result of the PID algorithm.

```
void calculatePID() {
  // adjust Kp to get the best results for the line follower
  float Kp = kp;

  // Calculate the error as the difference between the desired position (1000)
  // and the current position
  int16_t error = 1000 - currentPosition;

  // Set the motor speed based on proportional term.
  float pid = Kp * error;

  // Calculate the motor speeds based on the PID control signal
  int16_t motorLeftSpeed = constrain(baseSpeed - pid, minSpeed, maxSpeed);
  int16_t motorRightSpeed = constrain(baseSpeed + pid, minSpeed, maxSpeed);

  // Control the speed of both motors
  speedControl(motorLeftSpeed, motorLeftSpeed);
}
```

Obstacle avoidance

Upon detecting an obstacle:

- It positions itself with the left sensor above the line.
- Performs an arc using the "delay(ms)" function.
- Moves forward until it encounters the line and stops for 800ms.
- Resumes line following.



Communication between 2 microcontrollers

SerialTransfer

Using the PowerBroker2/SerialTransfer library, data packets are transmitted and received between two microcontrollers through the serial port (USART).

Transmit (Tx)

Data transmission is carried out using the TX (Transmit) port.

Each transmitted data packet is assigned an ID.

Callback

Upon receiving a data packet, the corresponding callback function is called based on the ID of the received data packet.

Receive (Rx)

The data type from the transmission is preserved.

The received data is decoded through the RX port in the order it was transmitted.

https://github.com/PowerBroker2/SerialTransfer

Transmit (Tx)

The data is initially stored in a buffer and then transmitted as a data packet with a unique ID. The variable "sendSize" is used to calculate the total size of the transmit buffer..

```
bool lineFollowerMode = true;
bool avoidObstacleMode = true;
int minSpeed = 0;
int baseSpeed = 255;
int maxSpeed = 200;
float kp = 0.40;

void transmitLineFollowerSettings() {
  // track of how many bytes we're stuffing in the transmit buffer
  uint16_t sendSize = 0;

  // Stuff buffer with data
  sendSize = serialTransfer.txObj(lineFollowerMode, sendSize);
  sendSize = serialTransfer.txObj(avoidObstacleMode, sendSize);
  sendSize = serialTransfer.txObj(baseSpeed, sendSize);
  sendSize = serialTransfer.txObj(maxSpeed, sendSize);
  sendSize = serialTransfer.txObj(minSpeed, sendSize);
  sendSize = serialTransfer.txObj(kp, sendSize);

  // Send buffer with packet ID
  serialTransfer.sendData(sendSize, PacketId::kLocalLineFollowerSettings);
}
```

Packet ID

The data packet ID is assigned using the enumeration (enum) called "PacketId."

```
enum PacketId {
  kLocalLineFollowerSettings,
  kCloudLineFollowerSettings,
  kESP32Status,
  kRemoteControlDirection,
};

```

Callback

The callback functions are indexed in an array of function pointers.
Upon receiving a data packet, the callback function with the index matching the ID of the received packet is called.

The "PacketId" enumeration allows the assignment of constants instead of numbers to define and identify the transmitted data packet IDs.

```
1  SerialTransfer serialTransfer;
2
3  // Callbacks.
4  // The order of the callbacks in the callbackArr array should match the order of
5  // the enum PacketId in the include/enum_packet_id.h file. This ensures that the
6  // correct callback function is called when a specific packet with a
7  // corresponding PacketId is received.
8  const functionPtr callbackArr[] = { receiveLineFollowerSettings };
9
10 /**
11  * @brief Config Serial Transfer
12  *
13  */
14 void setupSerialTransfer() {
15   Serial2.begin(9600, SERIAL_8N1, 16, 17);
16
17   // Config Serial Transfer
18   configST transferConfig;
19   transferConfig.debug = true;
20   transferConfig.callbacks = callbackArr;
21   transferConfig.callbacksLen = sizeof(callbackArr) / sizeof(functionPtr);
22
23   // Begin Serial Transfer
24   serialTransfer.begin(Serial2, transferConfig);
25 }
```

```
1  enum PacketId {
2    kLocalLineFollowerSettings,
3    kCloudLineFollowerSettings,
4    kESP32Status,
5    kRemoteControlDirection,
6  };
7
```

Receive (Rx)

Receiving the data must be done in exactly the same order as they were transmitted.

The variable "recSize" is used to calculate the size of the received data.

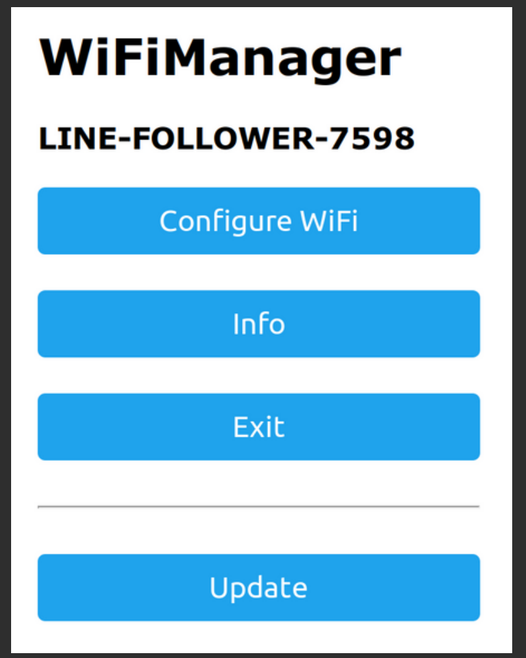
```
1  bool lineFollowerMode;
2  bool avoidObstacleMode;
3  int minSpeed;
4  int baseSpeed;
5  int maxSpeed;
6  float kp;
7
8  void receiveLineFollowerSettings() {
9    uint16_t recSize = 0;
10   recSize = serialTransfer.rxObj(lineFollowerMode, recSize);
11   recSize = serialTransfer.rxObj(avoidObstacleMode, recSize);
12   recSize = serialTransfer.rxObj(baseSpeed, recSize);
13   recSize = serialTransfer.rxObj(maxSpeed, recSize);
14   recSize = serialTransfer.rxObj(minSpeed, recSize);
15   recSize = serialTransfer.rxObj(kp, recSize);
16 }
```

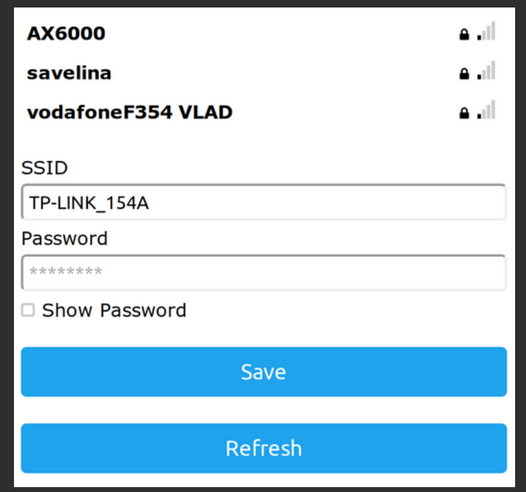
WiFi manager

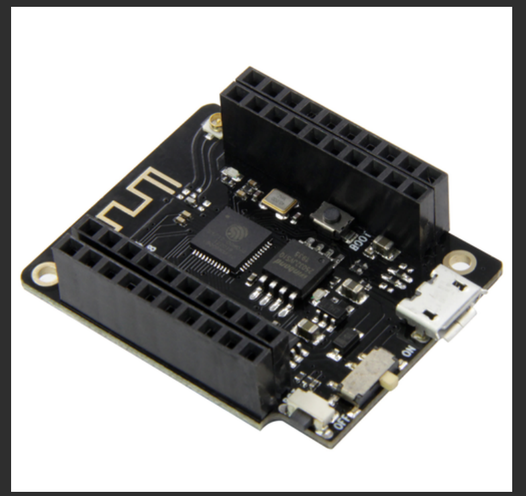
Connecting ESP32 to WiFi

Used library: tzapu/WiFiManager

https://github.com/tzapu/WiFiManager







WiFi portal setup example

The name of the WiFi network for the WiFi Manager portal is specified by the variable "hostname." Access is password-protected, assigned through the variable "password."

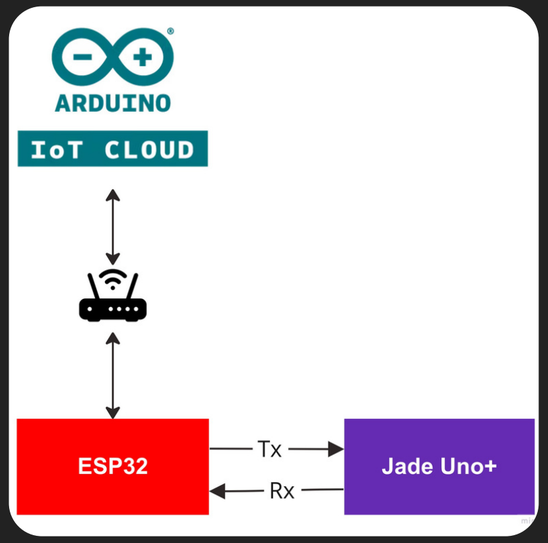
WiFi.mode(WIFI_STA) sets the mode to client/station.

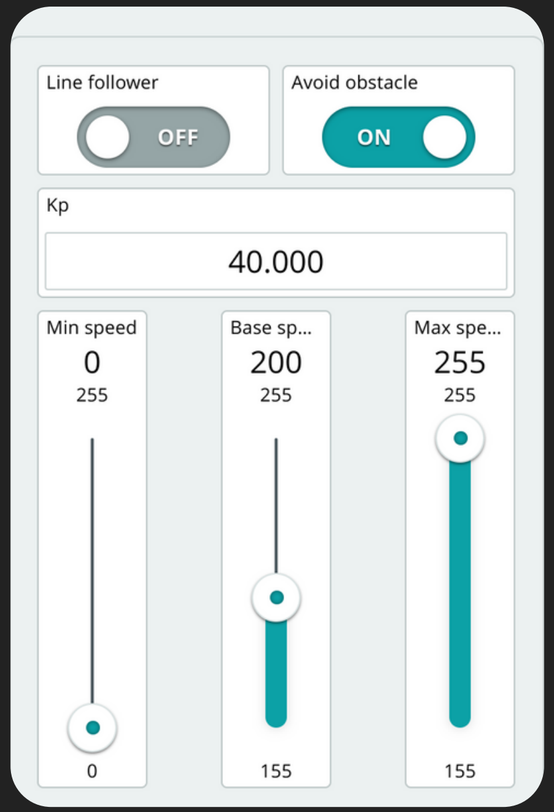
By using "wm.setConfigPortalBlocking(false)," the code runs in a non-blocking mode even if it hasn't successfully connected to the router. This allows data transmission between microcontrollers even if there's no connection to the router or the internet.

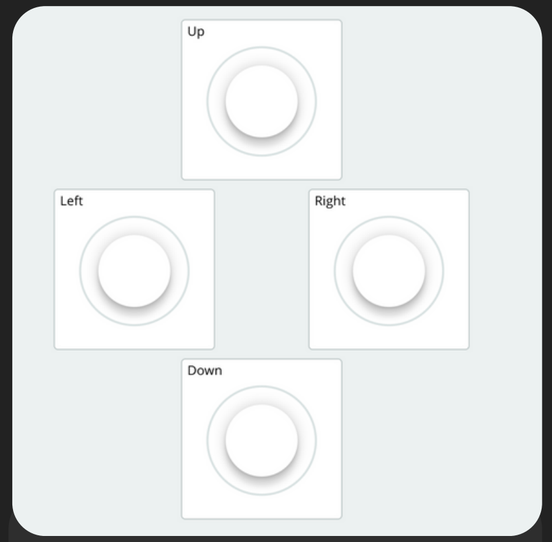
Calling "wm.process()" in the loop allows the code to run in a non-blocking mode, enabling it to execute other tasks as well.

```
WiFiManager wm;

char const hostname[] = "LINE-FOLLOWER-7598";
char const password[] = "password";

// Setup wifi manager
void setupWifiManager() {
  WiFi.mode(WIFI_STA);  // explicitly set mode, esp defaults to STA+AP
  // it is a good practice to make sure your code sets wifi mode how you want
  // it.

  // reset settings - wipe stored credentials for testing
  // these are stored by the esp library
  // wm.resetSettings();

  // Debug
  wm.setDebugOutput(false);

  // Set Hostname
  wm.setHostname(hostname);

  // Continue, even if not connected to WiFi
  wm.setConfigPortalBlocking(false);

  // Set a timeout period for the configuration portal
  wm.setConfigPortalTimeout(180);

  // Connect using saved credentials
  bool res;
  res = wm.autoConnect(hostname, password);  // password protected ap

  if (!res) {
    Serial.println("Failed to connect to wireless");
    // ESP.restart();
  } else {
    Serial.println("Connected to wireless");
  }
}

/**
 * @brief Loop Wifi Manager
 *
 */
void loopWiFiManager() {
  // Non blocking function for the WiFi manager
  wm.process();
}
```

Arduino cloud







Why Arduino cloud ?

- Widgets with a pleasant design
- Easy to configure
- Automatically generated code
- Mobile app
- Free account for 5 items

Dashboards

Two control panels have been created. One is for the line follower, and the other is for remote direction control.

https://cloud.arduino.cc

Configure Arduino cloud

Devices

The device registers on the Arduino Cloud platform from the Devices section.

Each device receives an ID and a secret key for identification.

Cloud Variables

Each variable is registered for every element on the control panel from the Things section.

Dashboards

From Dashboards, the widgets are added to the control panel.

For each widget, the corresponding variable is selected.

Sketch

The generated code is located in the Things > Sketch section.
The code can be uploaded directly from the browser or edited manually in the IDE editor.

https://cloud.arduino.cc

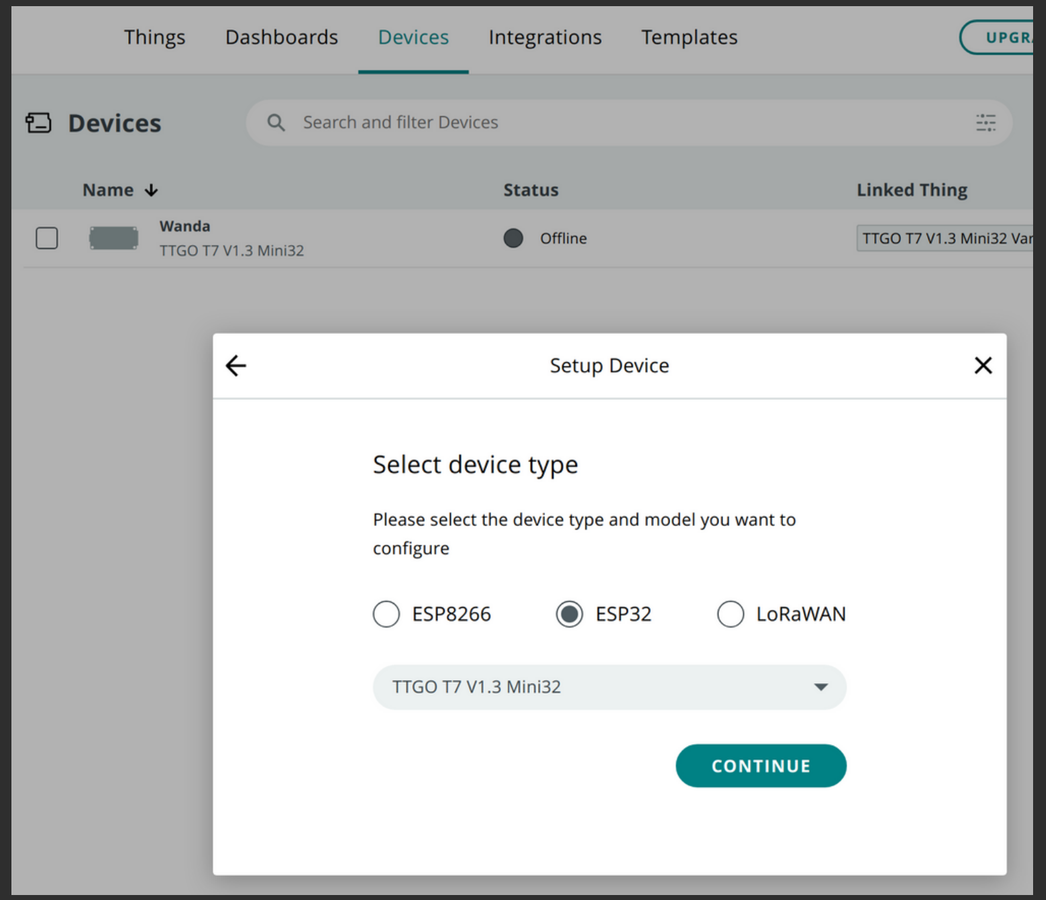
Arduino cloud

Devices

From the Devices section, the TTGO T7 V1.3 Mini32 device with the ESP32 microcontroller has been added to be integrated into Arduino Cloud.

After adding the device, the platform generated a unique ID and a secret key.

The ID and secret key are part of the code used for authentication with Arduino Cloud.



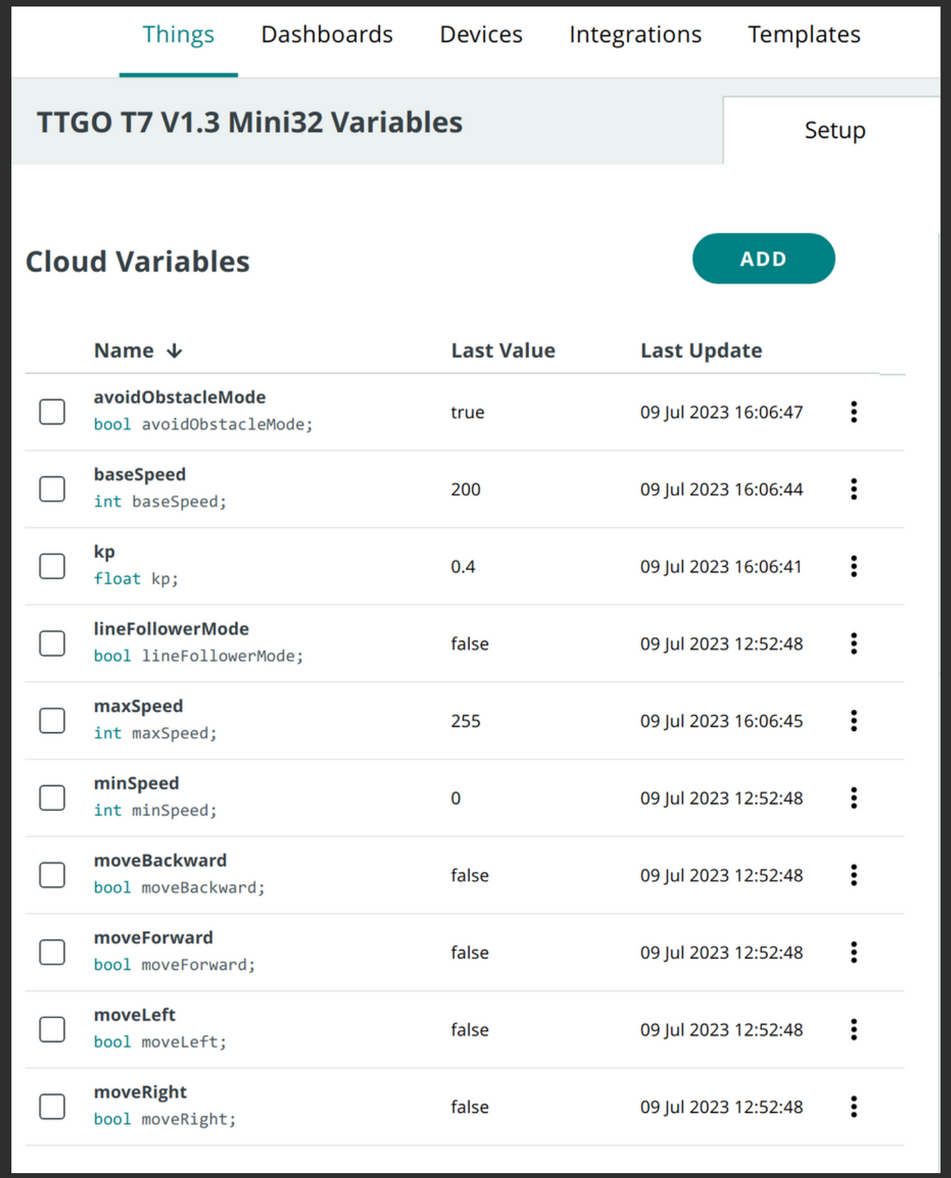
Arduino cloud

Cloud Variables

On the right side, the variables created for the line follower and remote control are presented.

Communication between microcontrollers

These variables are common to both microcontrollers and are defined in header files located in the "include/serial_transfer" directory. They are transmitted between microcontrollers through the serial port, using the SerialTransfer library.



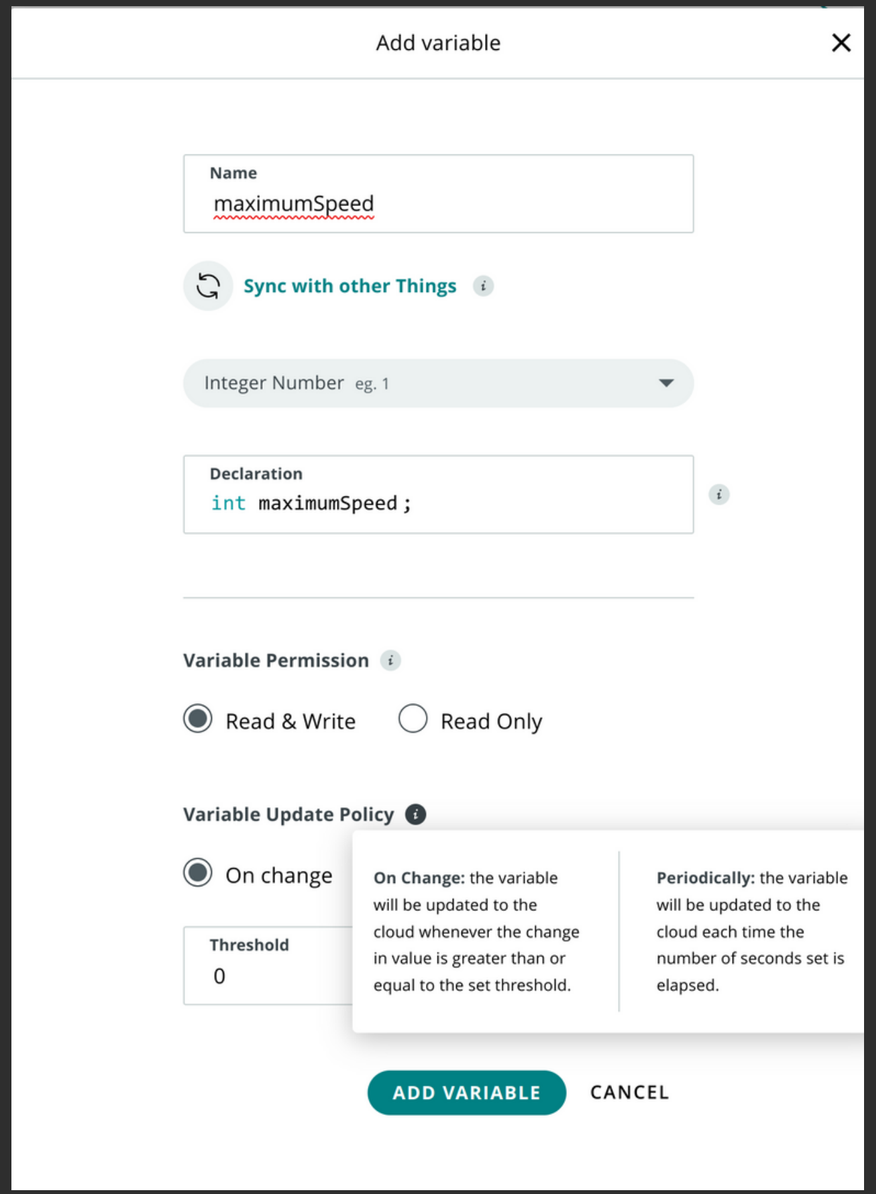
Arduino cloud

Adding Variables

From the "Things" > "Cloud Variables" section, you can add the variables you want to monitor or control via Arduino Cloud.

Fields to fill in:

- Variable name
- Variable type
- Edit option (whether it can be changed from the control panel or not)
- Update interval (when changed or at a specific time interval)



Arduino cloud

Dashboards

The accompanying image shows the widgets used to configure the line follower.

If Kp is set to zero, the line follower will not function because both wheels will rotate at the same speed regardless of the sensor's position.

Whenever any option is changed, a short beep from the buzzer will be heard to confirm the reception by the Jade Uno+ development board.

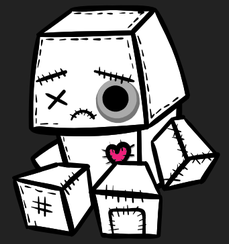
Arduino cloud

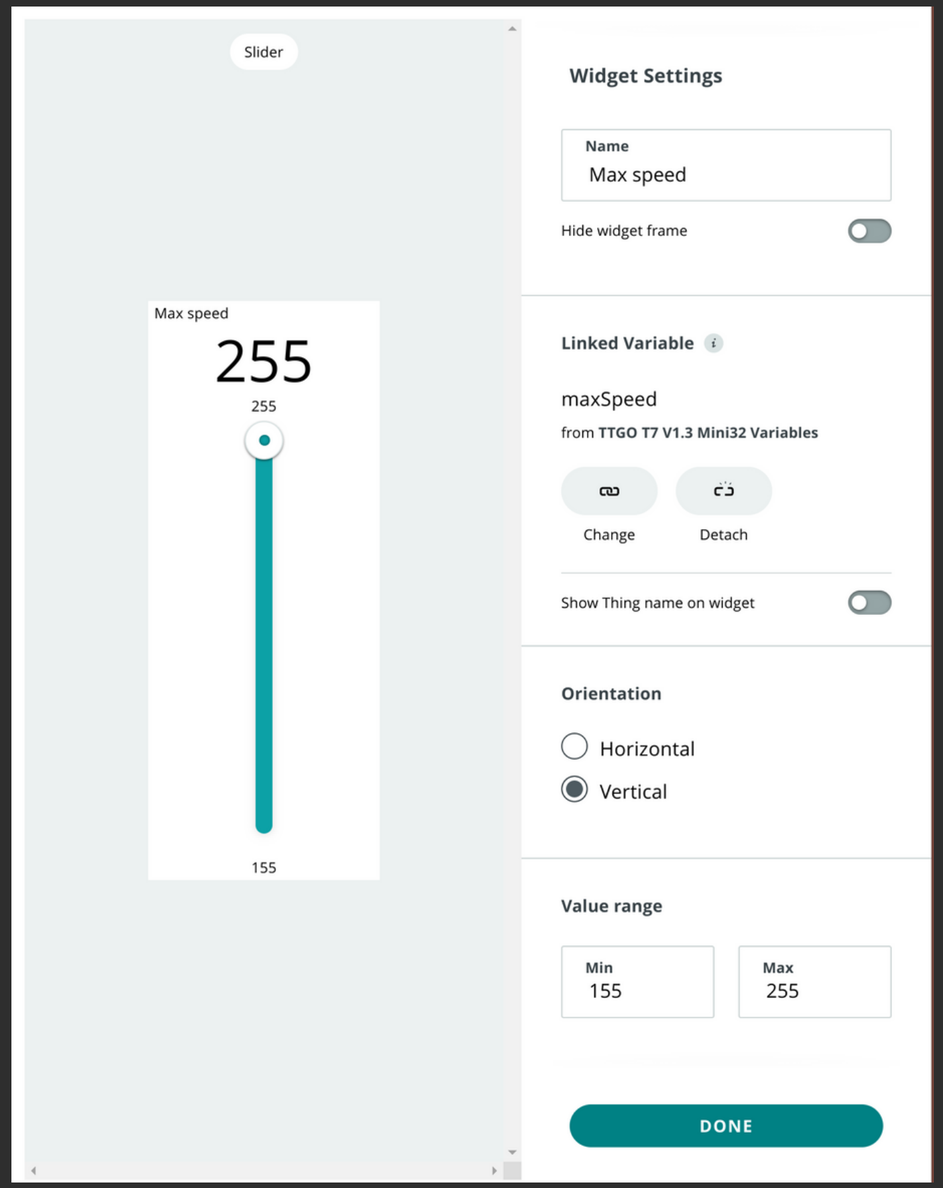
Editing a widget in the Dashboard

When editing the component, you need to link it to the variable that was previously created in "Things" > "Cloud Variables". This is done from the right-hand side, in the "Link Variable" section.

Widget limitations

The major disadvantage of Arduino Cloud is that it offers only 21 widgets that cannot be customized.





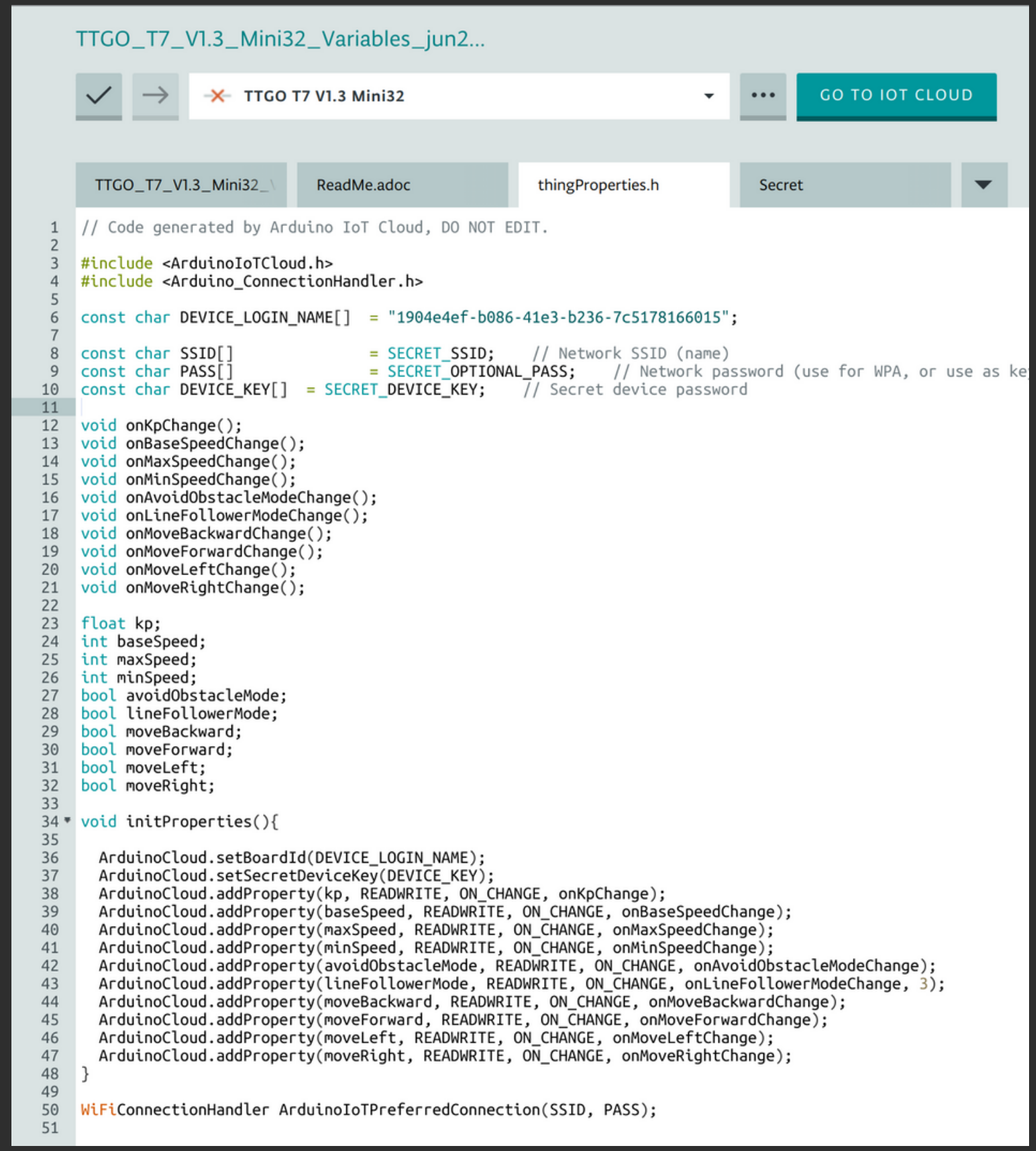
Arduino cloud

Sketch

After adding the widgets, the code is automatically generated and can be edited from the "Things" > "Sketch" section. From the Secret tab, you can modify the device ID, secret key, as well as the WiFi connection data for authentication in the cloud.

Connecting to WiFi

Arduino Cloud doesn't use WiFi manager by default, which is why it was necessary to integrate the WiFiManager library.



Arduino cloud

Other aspects

- Arduino Cloud only supports integer data types like int and unsigned int. Data types like uint8_t and uint16_t are not supported for transmission to the cloud by this library.
- The int data type has a size of 32 bits on ESP32 and 16 bits on ATmega328, which can lead to incorrect reception. To solve this issue, the solution involved using explicit casting to create variables of type uint8_t with the same value as the int variables. This casting ensures that the data is transmitted correctly to Arduino Cloud.

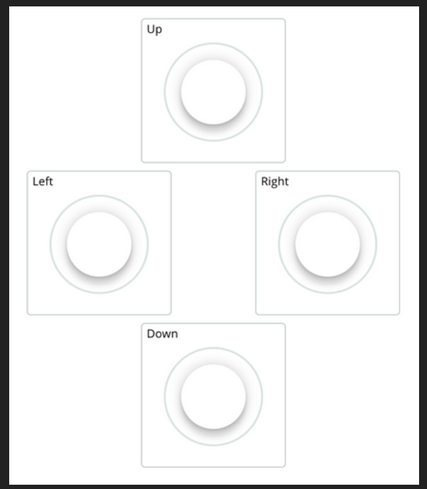
```
// The bellow code is adapted from the Arduino IoT Cloud - https://cloud.arduino.com

unsigned int maxSpeed;

/*
  Since MaxSpeed is READ_WRITE variable, onMaxSpeedChange() is
  executed every time a new value is received from IoT Cloud.
*/
void onMaxSpeedChange() {
  // Add your code here to act upon MaxSpeed change
  transmitLineFollowerSettings();
  Serial.print("onMaxSpeedChange maxSpeed ");
  Serial.println(maxSpeed);
}
/*
  On first Arduino Cloud Sync - synchronize local variables with cloud
  variables
*/
void onIoTSync() {
  ArduinoCloud.addProperty(maxSpeed, READWRITE, ON_CHANGE, onMaxSpeedChange);
}

/*
  Initialize Arduino Cloud
*/
void initThingProperties() {
  ArduinoCloud.setBoardId(DEVICE_LOGIN_NAME);
  ArduinoCloud.setSecretDeviceKey(DEVICE_KEY);

  // Request the lineFollowerSettings from Jade UnoPlus board
  requestLineFollowerSettings();

  // On first Arduino Cloud Sync - synchronize local variables with cloud
  // variables
  ArduinoCloud.addCallback(ArduinoIoTCloudEvent::SYNC, onIoTSync);
}

// Wifi manager extension handles the preferred connection method for Arduino
// IoT. This line does not affect the wifi connection.
WiFiConnectionHandler ArduinoIoTPreferredConnection("none", "none");
```

Arduino cloud

Remote control

Control Panel for Direction Control via Arduino Cloud.

Each button represents a push button.



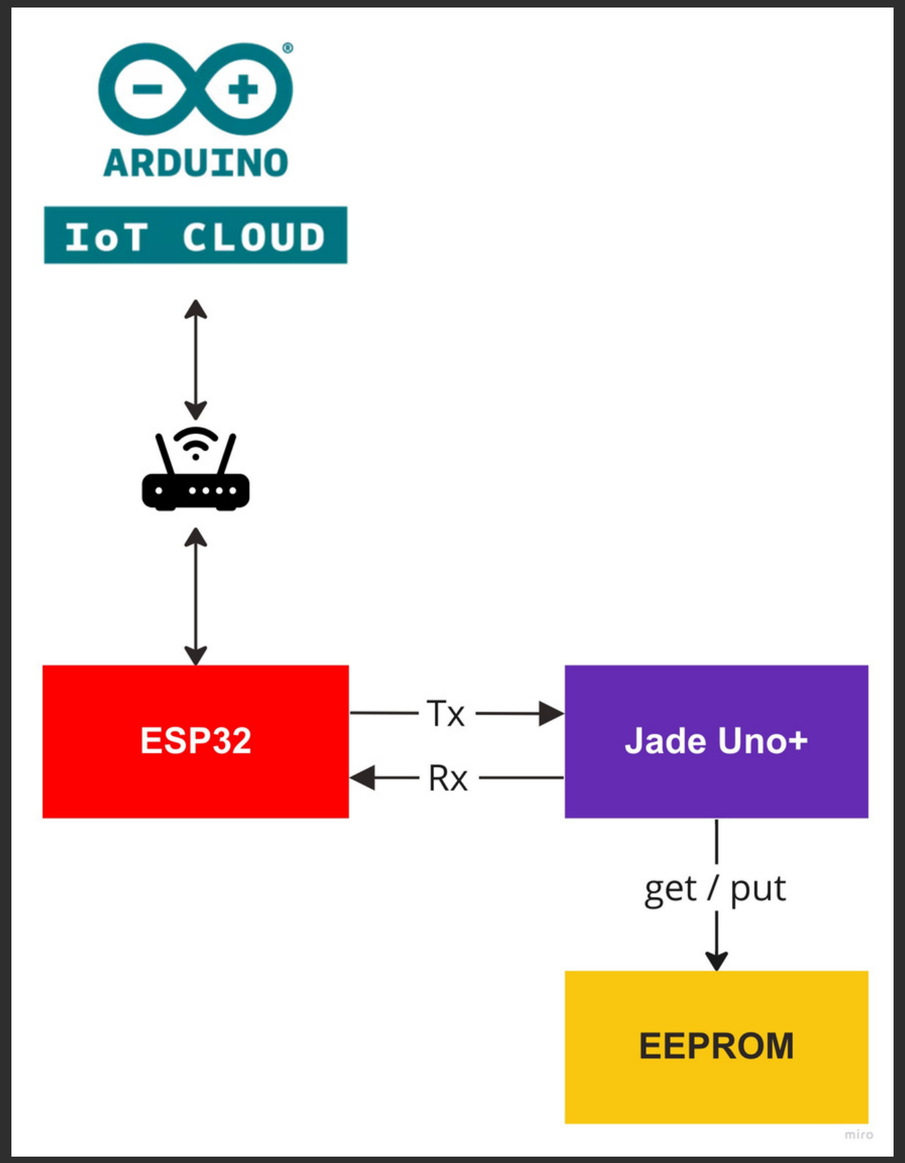

```
/*
  Since MoveLeft is READ_WRITE variable, onMoveLeftChange() is
  executed every time a new value is received from IoT Cloud.
*/
void onMoveLeftChange() {
  // Add your code here to act upon MoveLeft change
  if (moveLeft) {
    transmitDirection(LEFT_WIDE_FORWARD);
    return;
  }

  transmitDirection(STOP_SLOW);
}

/*
  Since MoveRight is READ_WRITE variable, onMoveRightChange() is
  executed every time a new value is received from IoT Cloud.
*/
void onMoveRightChange() {
  // Add your code here to act upon MoveRight change
  if (moveRight) {
    transmitDirection(RIGHT_WIDE_FORWARD);
    return;
  }

  transmitDirection(STOP_SLOW);
}


/*
  Initialize Arduino Cloud
*/
void initThingProperties() {
  ArduinoCloud.setBoardId(DEVICE_LOGIN_NAME);
  ArduinoCloud.setSecretDeviceKey(DEVICE_KEY);

  // Remote control
  ArduinoCloud.addProperty(moveBackward, READWRITE, ON_CHANGE, onMoveBackwardChange);
  ArduinoCloud.addProperty(moveForward, READWRITE, ON_CHANGE, onMoveForwardChange);
  ArduinoCloud.addProperty(moveLeft, READWRITE, ON_CHANGE, onMoveLeftChange);
  ArduinoCloud.addProperty(moveRight, READWRITE, ON_CHANGE, onMoveRightChange);
}

WiFiConnectionHandler ArduinoIoTPreferredConnection("none", "none");
```

EEPROM

Why EEPROM?

Arduino Cloud allows modifying the values of components in the control panel even when the device is turned off or does not have an active internet connection.

To ensure that the device uses the latest configurations used before shutdown, and not the ones modified in the cloud during the inactive period, a 1kB persistent EEPROM memory is used. This way, upon restarting the device, it will load the locally saved settings from EEPROM, not the ones from the cloud.



EEPROM

Initialization

When powering up the Jade Uno+ board, if the EEPROM memory is empty, the default settings are saved.

Data retrieval

Using the EEPROM.get() method assigns values stored in the EEPROM memory to global variables representing the settings of the line follower.

Saving data

When modifying data in Arduino Cloud, these changes are saved in the persistent EEPROM memory.

```
/**
 * @brief Initialize EEPROM
 */
void initializeEEPROM() {
  // Read the first byte from EEPROM
  byte firstByte = EEPROM.read(0);

  if (firstByte == 0xFF) {
    // Initialize the line follower settings with default settings
    initializeLineFollowerSettings();

    // Save the line follower settings to EEPROM
    saveLineFollowerSettings();
  }

  // Get EEPROM data
  getEEPROMData();
}

/**
 * @brief Get EEPROM data
 *
 */
void getEEPROMData() {
  EEPROM.get(0, avoidObstacleMode);
  EEPROM.get(1, minSpeed);
  EEPROM.get(2, baseSpeed);
  EEPROM.get(3, maxSpeed);
  EEPROM.get(4, kp);
}

/**
 * @brief Save lineFollowerSettings to EEPROM
 *
 */
void saveLineFollowerSettings() {
  EEPROM.put(0, avoidObstacleMode);
  EEPROM.put(1, minSpeed);
  EEPROM.put(2, baseSpeed);
  EEPROM.put(3, maxSpeed);
  EEPROM.put(4, kp);

  Serial.println("Save the line follower settings to EEPROM");
}
```

LCD display

RSSI

The signal strength is displayed on the LCD in dBm (decibels-milliwatts). A good signal is considered to be around -50 dBm. A signal of -90 dBm is considered unusable.

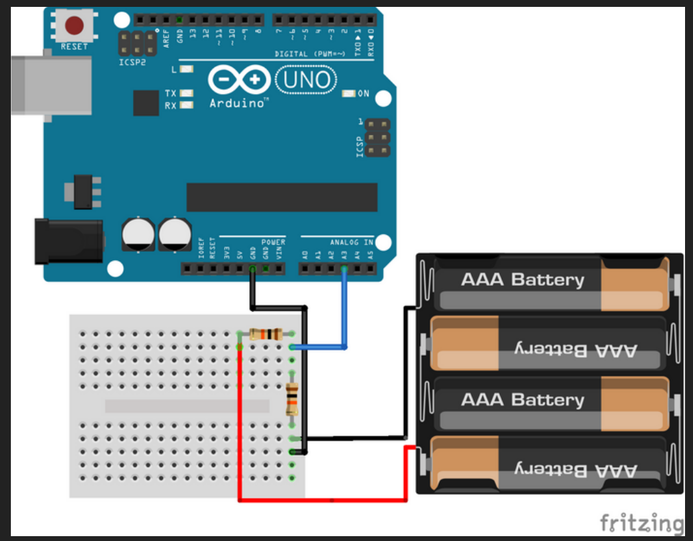
Cloud

The cloud connection is displayed on the right side using two special characters, each formed by a matrix of 5x8 pixels.

Battery voltage

On the second row, the voltage for the 18650 battery module and the voltage for the AA batteries (1.2V x 4) are displayed.

Voltmeter



Measuring AA batteries voltage.

To measure the voltage of the four AA batteries, the voltage divider method was used.

$$V_{\text{out}} = \frac{R_2}{R_1 + R_2} V_{\text{in}}$$

The variable "boardVoltage" represents the voltage measured between the development board's pins (5V and GND).
R1 is the resistor connected to the positive terminal of the batteries, and R2 is the resistor connected to ground.
To calculate the output voltage (Vout) across the two resistors, the formula used is Vout = (analogValue * boardVoltage) / 1024.0.

```
float readAAbatteryVoltage() {
  float Vin = 0.0;
  float Vout = 0.0;
  float r1 = 9810.0;
  float r2 = 9780.0;
  float boardVoltage = 4.77;

  // Read the AA voltage
  uint16_t analogValue = analogRead(AA_BATTERY_VOLTAGE_PIN);

  // Conversion formula
  Vout = (analogValue * boardVoltage) / 1024.0;
  Vin = Vout / (r2 / (r1 + r2));
  if (Vin < 0.1) {
    Vin = 0.0;
  }

  return Vin;
}
```

RGB led

System status

Meaning of colors:

WiFi disconnected (intermittent blue)

Arduino Cloud disconnected (intermittent purple)
Arduino Cloud connected (continuous purple)

Line follower active (continuous green)

Line follower stopped (continuous red)

```cpp

bool ledBlink = false;
uint8_t ledColorRed = 0;
uint8_t ledColorGreen = 0;
uint8_t ledColorBlue = 0;

/**
 * @brief Multicolor LED loop called in the main loop
 *
 */
void loopMultiColorLed() {
  toggleLedBlink();
}

/**
 * @brief Change multicolor led pins values
 *
 * This function sets the color of a RGB LED by adjusting the intensity of its red, green, and blue channels.
 *
 * @param r The intensity of the red channel (0-255).
 * @param g The intensity of the red channel (0-255).
 * @param b The intensity of the red channel (0-255).
 */
void changePINvalue(uint8_t r, uint8_t g, uint8_t b) {
  analogWrite(RED_PIN, r);
  analogWrite(GREEN_PIN, g);
  analogWrite(BLUE_PIN, b);
}

/**
 * @brief Set the multicolor led variables
 *
 * @param r The intensity of the red channel (0-255).
 * @param g The intensity of the red channel (0-255).
 * @param b The intensity of the red channel (0-255).
 * @param blink Blink (true or false)
 */
void setMultiColorLed(uint8_t r, uint8_t g, uint8_t b, bool blink) {
  ledColorRed = r;
  ledColorGreen = g;
  ledColorBlue = b;
  ledBlink = blink;

  changePINvalue(r, g, b);
}

/*
* @brief Toggle the multicolor led blink
*
*/
void toggleLedBlink() {
  if (ledBlink == false) {
    return;
  }

  unsigned long currentTime = millis();
  static unsigned long startTime = currentTime;
  static bool isLedOn = false;

  if (currentTime - startTime >= 600) {

    isLedOn = !isLedOn;
    // Turn off the LED for 600ms
    if (isLedOn) {
      changePINvalue(LOW, LOW, LOW);
    } else {
      changePINvalue(ledColorRed, ledColorGreen, ledColorBlue);
    }

    startTime = currentTime;
  }
}

```

Buzzer

Wire connection

The buzzer module used emits sounds when the I/O pin receives a LOW signal.

To use the tone() function in Arduino, the behavior of the buzzer needs to be inverted so that it emits sounds when it receives a HIGH signal. This can be achieved by connecting the I/O pin to GND and providing the digital signal directly to the VCC pin of the buzzer. Consequently, the buzzer will emit sounds when the signal is HIGH.

```cpp
/**
 * @brief Play a tone
 *
 * @param noteFrequency The frequency of the tone to play
 * @param noteDuration The duration of the tone to play
 * @param silentDuration The duration of the silence after the tone
 */
void playTone(float noteFrequency, long noteDuration, int silentDuration) {
  if (silentDuration == 0) {
    silentDuration = 1;
  }
  tone(BUZZER_PIN, noteFrequency, noteDuration);
  delay(silentDuration);
}

/**
 * @brief Play a tone with a frequency that changes from initFrequency to finalFrequency
 *
 * @param initFrequency Initial frequency
 * @param finalFrequency Final frequency
 * @param prop Proportion of the frequency change
 * @param noteDuration Duration of the tone
 * @param silentDuration Duration of the silence after the tone
 */
void bendTones(float initFrequency, float finalFrequency, float prop,
               long noteDuration, int silentDuration) {
  if (silentDuration == 0) {
    silentDuration = 1;
  }

  float currentFrequency = initFrequency;
  if (initFrequency < finalFrequency) {
    while (currentFrequency < finalFrequency) {
      playTone(currentFrequency, noteDuration, silentDuration);
      currentFrequency *= prop;
    }
  } else {
    while (currentFrequency > finalFrequency) {
      playTone(currentFrequency, noteDuration, silentDuration);
      currentFrequency /= prop;
    }
  }
  noTone(BUZZER_PIN);
}

/**
 * @brief Play a sequence of tones
 *
 * @param sequence The sequence to play
 */
void playSequence(uint8_t sequence) {
  switch (sequence) {
    case S_CONNECTION:
      playTone(NOTE_E5, 50, 30);
      playTone(NOTE_E6, 55, 25);
      playTone(NOTE_A6, 60, 10);
      break;
    case S_BUTTON_PUSHED:
      bendTones(NOTE_E6, NOTE_D7, 1.04, 10, 2);
    break;
    case S_STOP:
      bendTones(100, 500, 1.04, 10, 20);
      delay(500);
      bendTones(400, 100, 1.04, 10, 11);
      break;
    case S_OBSTACLE:
      bendTones(1000, 1700, 1.03, 8, 10);
      bendTones(1699, 500, 1.04, 8, 11);
      bendTones(1000, 1700, 1.05, 9, 19);
      break;
  }
}
```