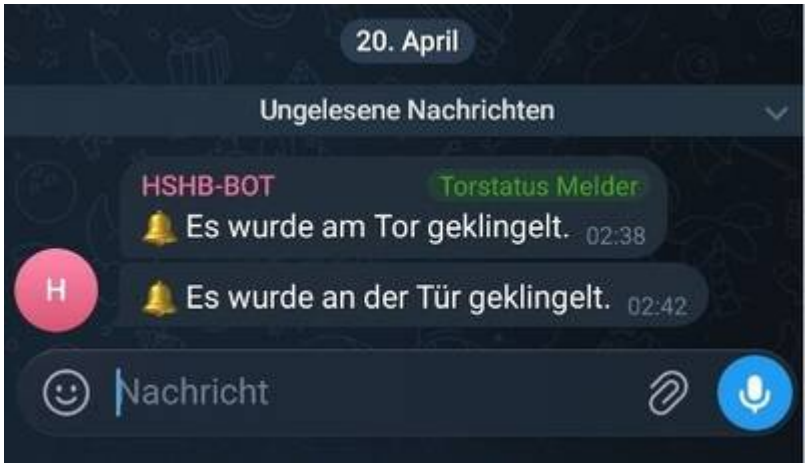
Klingeldetektor

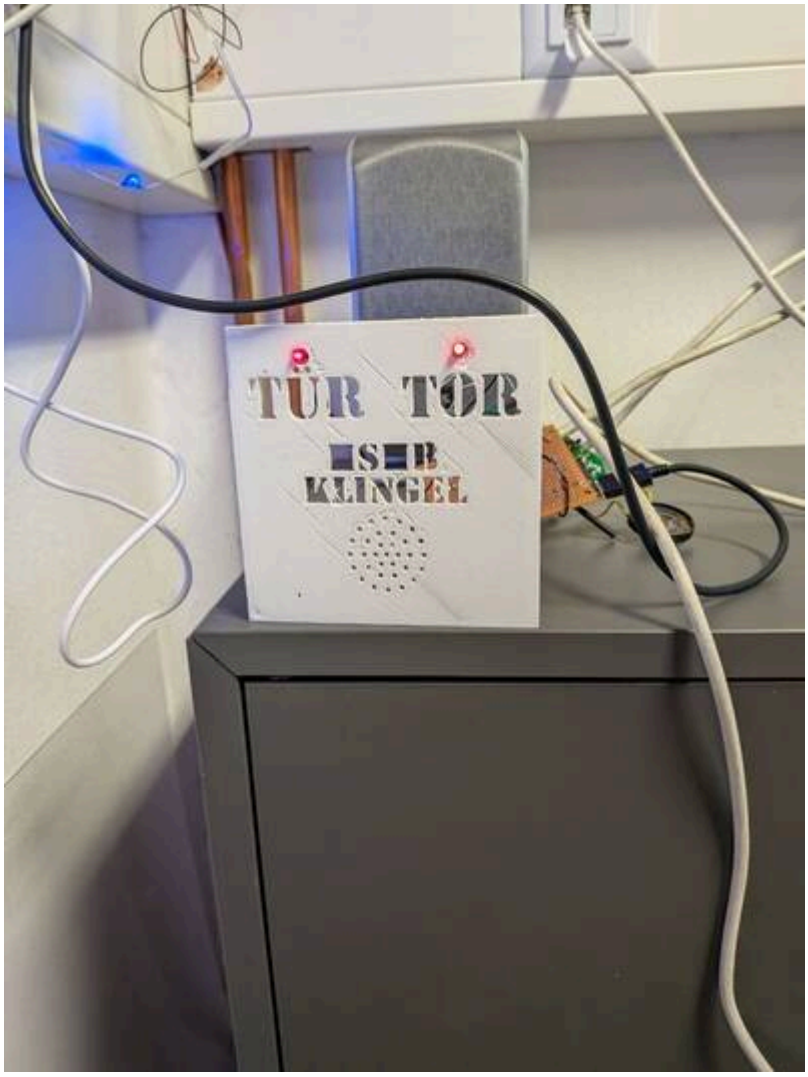


Zurzeit ist die Funkbrücke in die E-Werkstatt und den Kreativraum außer Betrieb! Auch wird die Torklingel zurzeit überarbeitet!

Es wird unsere Telegram Gruppe gepostet sobald geklingelt wurde.



HSHB Telegram Gruppe



Der Klingeldetektor erkennt, ob an der Tür oder am Tor geklingelt wurde.

Erkennungswege

- **Tür:** Auswertung des Datenbusses des hauseigenen Klingelsystems (**QwikBus**)
- **Tor:** Erkennung über **LDR (Lichtsensor)** am Empfänger der Torklingel

Bei einem Klingelereignis

- LED-Visualisierung Ort (**Tor / Tür**)
- Akustische und visuelle Anzeige im Hauptraum
- Klingelsignal weitergabe an **E-Werkstatt** und **Kreativwerkstatt** (über eine **Funkbrücke**)
- Meldung an Telegram (Gruppe)
- Meldung per **MQTT** im Hackerspace-System (für zukünftige Erweiterungen)

Verhalten im Standby

- Warten auf ein Klingelereignis

- LEDs werden ein- und ausgeblendet, um Standby und Funktionsbereitschaft zu visualisieren

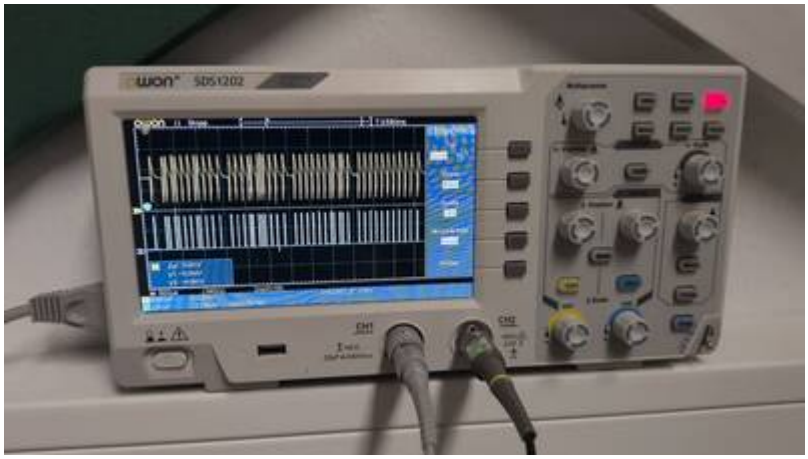
Umgesetzte Extras

- MQTT-Integration
- Akustische und visuelle Anzeige im Hauptraum
- Funkverteilung in die anderen Räume
- LED-Visualisierung (Tor / Bornstraße)

Danksagungen:

- **Marco** – Tips bei der Entwicklung der Hardware
- **Delpiño** – Unterstützung beim Auswertendes Datenbusses
- **Markus** – Idee zur LED-Visualisierung (Tor / Bornstraße)

Technische Erklärung



Tür – QwikBus-Auswertung

Das hauseigene Klingelsystem (QwikBus) überträgt Klingelereignisse als zeitkodierte elektrische Signale.

Signalcharakteristik:

- Ruhespannung: ca. ~20 V
- Aktivsignal: ca. ~18 V
- Information in **zeitlichen Pulsfolgen**

Hardware:

- 1 nF um nur die Busänderungen zu dedektieren
- Spannungsteiler
- Schmitt-Trigger

- Mikrocontroller (ESP32)

Verarbeitung:

- Flankenerkennung (Interrupt)
- Zeitmessung im µs-Bereich
- Bildung von Sequenzen:
- Quantisierung zur Reduktion von Rauschen
- Mustervergleich (Pattern Matching) zur Klassifikation

Ergebnis:

- Zuordnung des Signals zu einem **Türklingel-Ereignis**

Tor – LDR-Auswertung

Das Torklingeln wird über das optische Signal des Empfängers erkannt.

Signalquelle:

- Blinkende LED am Empfänger der Torklingel

Hardware:

- LDR (Lichtsensor)
- Widerstand als Spannungsteiler
- ADC-Eingang am Mikrocontroller

Verarbeitung:

- Kontinuierliche Helligkeitsmessung
- Schwellwertbasierte Erkennung (Hell/Dunkel)
- Zeitliche Analyse des Blinkmusters
- Entprellung zur Vermeidung von Fehltriggern

Ergebnis:

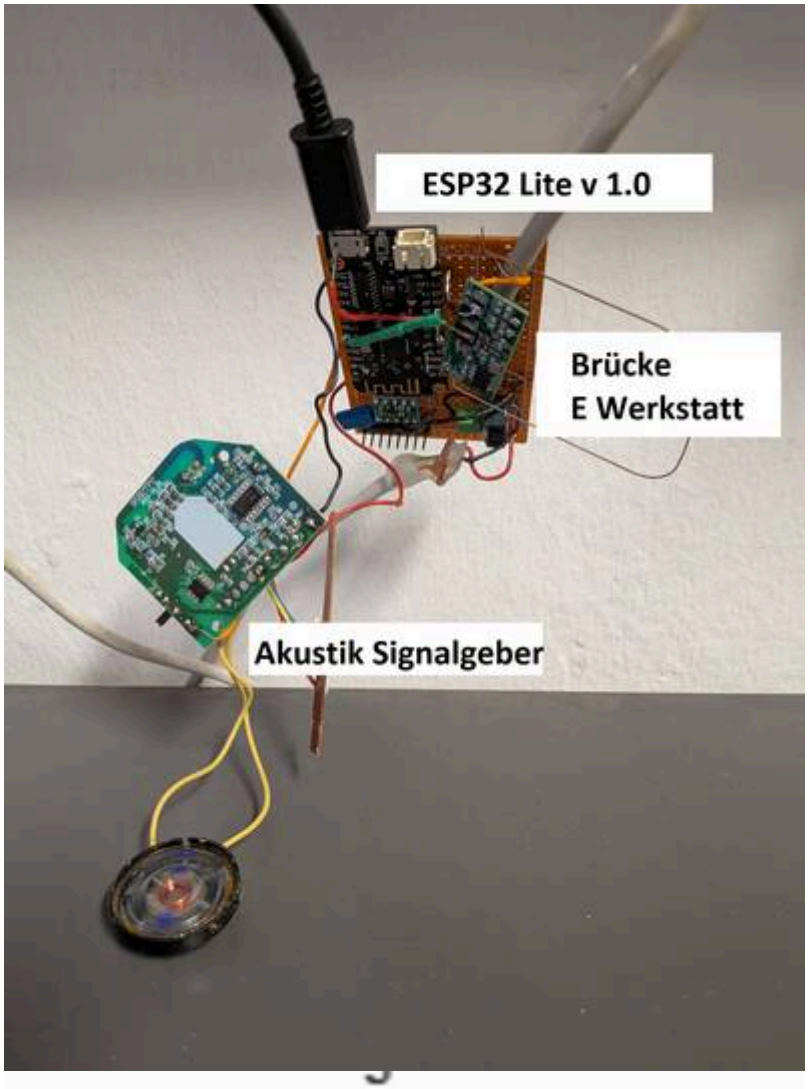
- Zuordnung des Signals zu einem **Torklingel-Ereignis**

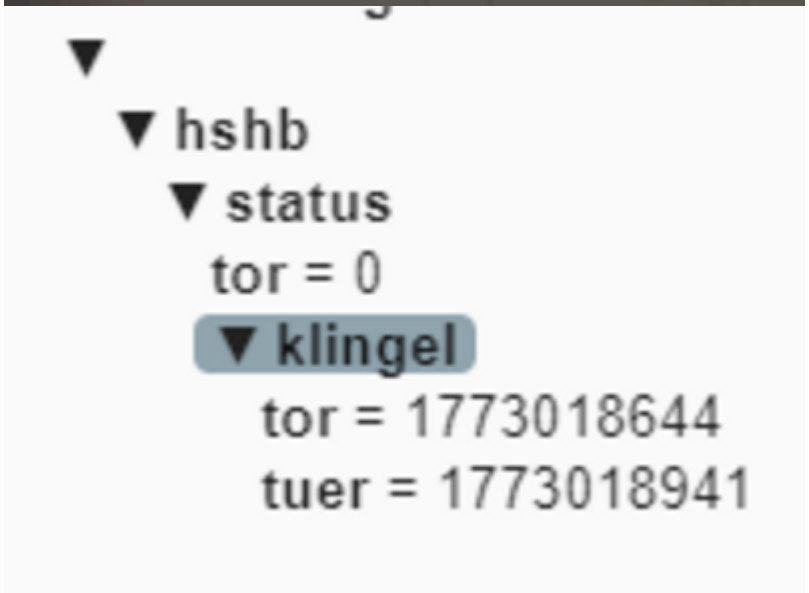
Ereignisverarbeitung

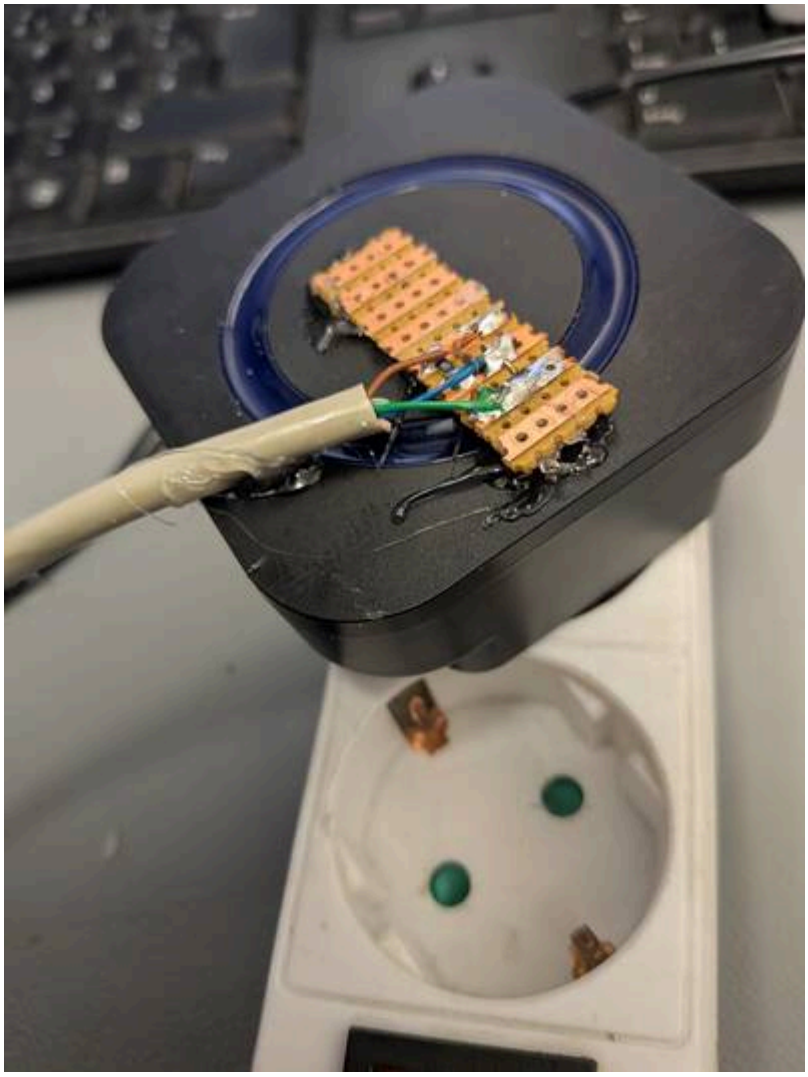
Nach der Erkennung erfolgt die weitere Verarbeitung:

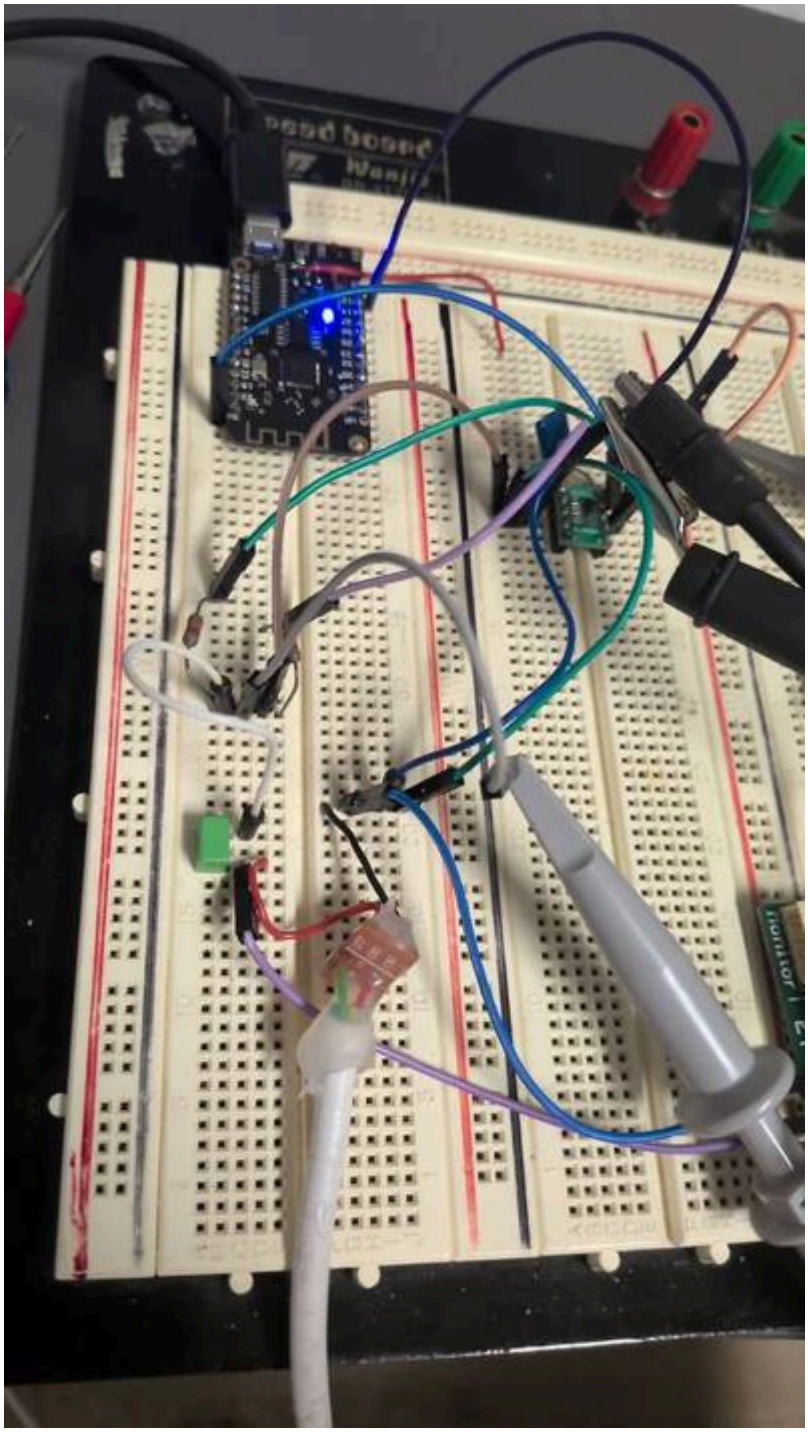
- Klassifikation: **Tür** oder **Tor**
- Generierung eines Events
- Veröffentlichung per **MQTT** im Hackerspace-System
- Ansteuerung der **LED-Visualisierung (Tor / Bornstraße)**
- Weitergabe der Information:
    - Akustisch und visuell im **Hauptraum**

- Über **Funkbrücke** an **E-Werkstatt** und **Kreativwerkstatt**









Code

```
#include <WiFi.h>
#include <esp_wifi.h>
#include <PubSubClient.h>
#include <time.h>
#include <Preferences.h>
#include <WebServer.h>
#include <HTTPUpdateServer.h>
#include <ArduinoOTA.h>
#include <ESPmDNS.h>
#include <Arduino.h>
```

```
#include <cstring>
#include "esp32-hal-rmt.h"
#include <math.h>

// ================================================
// WLAN / MQTT
// ================================================
const char* WIFI_SSID     = "Hackerspace";
const char* WIFI_PASSWORD = "****";

const char* MQTT_HOST = "daniel-stingl.de";
const int   MQTT_PORT = ****;
const char* MQTT_USER = "***";
const char* MQTT_PASS = "***";

const char* MQTT_TOPIC_TUER      = "/hshb/status/klingel/tuer";
const char* MQTT_TOPIC_TOR       = "/hshb/status/klingel/tor";
const char* MQTT_TOPIC_STATUS    = "/hshb/status/klingel/status";
const char* MQTT_TOPIC_TUER_TEST = "/hshb/status/klingel/tuertest";
const char* MQTT_TOPIC_TOR_TEST  = "/hshb/status/klingel/tortest";

// ================================================
// OTA / Web
// ================================================
const char* OTA_HOSTNAME = "hshb-klingel";
const char* OTA_USER     = "****";
const char* OTA_PASS     = "****";

// ================================================
// NTP
// ================================================
const char* NTP_SERVER = "pool.ntp.org";
const long  GMT_OFFSET_SEC = 3600;
const int   DAYLIGHT_OFFSET_SEC = 3600;

// ================================================
// MAC
// ================================================
uint8_t newMACAddress[] = {0x30, 0xAF, 0xA0, 0x05, 0x1D, 0x63};

// ================================================
// Pins
// ================================================
constexpr int bellPin    = 5;   // Trigger-Ausgang
constexpr int busPin     = 32;  // RMT RX
const int adcTorPin      = 33;  // ADC
const int testBellPin    = 26;  // Taster

// ================================================
// RMT / Nachrichten
// ================================================
```

```
constexpr uint8_t hackerspaceBell[]  = {0x25, 0x2E, 0xC4, 0x00, 0xE8};
constexpr uint8_t hackerspaceBell2[] = {0x25, 0x2E, 0xE2, 0x00, 0xCA};

constexpr size_t bufferSize = 16;
constexpr size_t maxSyms = RMT_MEM_NUM_BLOCKS_2 *
RMT_SYMBOLS_PER_CHANNEL_BLOCK;

rmt_data_t symbols[maxSyms];
size_t symbolCount = maxSyms;

enum { P_BAD = -1, P_ZERO = 0, P_ONE = 1, P_START = 2, P_STOP = 3 };

// ====================================================
// Zeiten / Zustände
// ====================================================
unsigned long lastBellTriggered       = 0;
unsigned long lastTorTriggerMillis    = 0;
unsigned long lastButtonTriggerMillis = 0;
unsigned long lastDebugMillis         = 0;
unsigned long lastStatusMillis        = 0;
unsigned long bellPulseStartMillis    = 0;
unsigned long lastWifiReconnectMs     = 0;
unsigned long lastMqttReconnectMs     = 0;
unsigned long lastNtpSyncMs           = 0;

const unsigned long BellDebounceMs     = 300;
const unsigned long TorDebounceMs      = 11000;
const unsigned long ButtonDebounceMs   = 300;
const unsigned long Debuginterval      = 1000;
const unsigned long StatusMs           = 5000;
const unsigned long BellPulseMs        = 10;
const unsigned long WifiRetryMs        = 10000;
const unsigned long MqttRetryMs        = 5000;
const unsigned long NtpResyncMs        = 6UL * 60UL * 60UL * 1000UL;
const unsigned long LedEventDurationMs = 60000;  // 60 Sekunden

const int adcTorThreshold = 1500;

bool bellPulseActive = false;
bool timeValid = false;
bool lastButtonState = HIGH;
bool lastTorState = false;

String lastState = "IDLE";
String lastSource = "-";
String lastDecodedHex = "-";
uint32_t lastDecodedLen = 0;
int adcTorValue = 0;

// ========================
// LEDs
```

```
// ========================
const int ledTuerPin = 23;
const int ledTorPin  = 19;

const int ledFreq = 5000;
const int ledResolution = 8;

enum LedMode {
  LED_MODE_STANDBY,
  LED_MODE_SHOW_TUER,
  LED_MODE_SHOW_TOR
};

LedMode ledMode = LED_MODE_STANDBY;
unsigned long ledModeUntilMs = 0;
unsigned long lastLedUpdateMs = 0;
unsigned long lastBlinkToggleMs = 0;
bool ledBlinkState = false;

// ======================================================
// Objekte
// ======================================================
WiFiClient wifiClient;
PubSubClient mqttClient(wifiClient);
Preferences preferences;
WebServer httpServer(80);
HTTPUpdateServer httpUpdater;

// ======================================================
// Vorwärtsdeklarationen
// ======================================================
bool connectWiFi();
bool ensureMqttConnected();
bool syncTime();
bool ensureTimeValid();
bool publishUnixTimeToTopic(const char* topic);
bool publishStatus();
void markMqttPendingTuer();
void markMqttPendingTor();
bool isMqttPendingTuer();
bool isMqttPendingTor();
void clearMqttPendingTuer();
void clearMqttPendingTor();
void processPendingMqtt();
void setupHttpRoutes();
void setupArduinoOta();
void triggerBell(bool fromButton);
void triggerTorEvent(const char* source);
void mqttCallback(char* topic, byte* payload, unsigned int length);
String buildDebugJson();
```

```
int pulseType(uint16_t us);
size_t collectLows(const rmt_data_t *symbols, size_t symbolCount, uint16_t
*lows);
int decodeByte(const uint16_t *pulses);
bool decodeMessage(const rmt_data_t *symbols, size_t symbolCount, uint8_t
*msg, size_t &len);
bool messageContains(const uint8_t *msg, size_t len, const uint8_t *what,
size_t whatLen);

// =====================================================
// WLAN / Zeit / MQTT
// =====================================================
bool connectWiFi() {
  if (WiFi.status() == WL_CONNECTED) return true;

  unsigned long now = millis();
  if (now - lastWifiReconnectMs < WifiRetryMs) return false;
  lastWifiReconnectMs = now;

  Serial.println("Verbinde mit WLAN...");
  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

  unsigned long startAttemptTime = millis();
  while (WiFi.status() != WL_CONNECTED && millis() - startAttemptTime <
10000) {
    delay(250);
    Serial.print(".");
  }
  Serial.println();

  if (WiFi.status() == WL_CONNECTED) {
    Serial.println("WLAN verbunden");
    Serial.print("IP: ");
    Serial.println(WiFi.localIP());
    return true;
  }

  Serial.println("WLAN Verbindung fehlgeschlagen");
  return false;
}

bool syncTime() {
  if (WiFi.status() != WL_CONNECTED) return false;

  Serial.println("Synchronisiere Zeit per NTP...");
  configTime(GMT_OFFSET_SEC, DAYLIGHT_OFFSET_SEC, NTP_SERVER);

  time_t now = time(nullptr);
  int retry = 0;
```

```
  while (now < 100000 && retry < 20) {
    delay(500);
    Serial.print(".");
    now = time(nullptr);
    retry++;
  }
  Serial.println();

  if (now < 100000) {
    Serial.println("NTP Zeit konnte nicht geholt werden");
    timeValid = false;
    return false;
  }

  Serial.print("Aktuelle Unix-Zeit: ");
  Serial.println((unsigned long)now);
  lastNtpSyncMs = millis();
  timeValid = true;
  return true;
}

bool ensureTimeValid() {
  time_t now = time(nullptr);
  if (now > 100000 && timeValid) {
    if (millis() - lastNtpSyncMs < NtpResyncMs) {
      return true;
    }
  }
  return syncTime();
}

bool ensureMqttConnected() {
  mqttClient.setServer(MQTT_HOST, MQTT_PORT);

  if (mqttClient.connected()) return true;
  if (WiFi.status() != WL_CONNECTED) return false;

  unsigned long now = millis();
  if (now - lastMqttReconnectMs < MqttRetryMs) return false;
  lastMqttReconnectMs = now;

  String clientId = "esp32-klingel-";
  clientId += String((uint32_t)ESP.getEfuseMac(), HEX);

  Serial.println("Verbinde MQTT...");
  if (mqttClient.connect(clientId.c_str(), MQTT_USER, MQTT_PASS)) {
    Serial.println("MQTT verbunden");

    mqttClient.subscribe(MQTT_TOPIC_TUER_TEST);
    mqttClient.subscribe(MQTT_TOPIC_TOR_TEST);
    Serial.println("MQTT Test-Topics subscribed");
```

```
    return true;
  }

  Serial.print("MQTT Verbindung fehlgeschlagen, rc=");
  Serial.println(mqttClient.state());
  return false;
}

bool publishUnixTimeToTopic(const char* topic) {
  if (!connectWiFi()) return false;
  if (!ensureTimeValid()) return false;
  if (!ensureMqttConnected()) return false;

  time_t now = time(nullptr);
  uint64_t nowMs = ((uint64_t)now) * 1000ULL + (millis() % 1000ULL);

  char payload[32];
  snprintf(payload, sizeof(payload), "%llu", (unsigned long long)nowMs);

  bool ok = mqttClient.publish(topic, payload, true);

  if (ok) {
    Serial.print("MQTT gesendet an ");
    Serial.print(topic);
    Serial.print(" mit Payload: ");
    Serial.println(payload);
  } else {
    Serial.print("MQTT publish fehlgeschlagen für ");
    Serial.println(topic);
  }

  mqttClient.loop();
  return ok;
}

void updateLeds() {
  unsigned long now = millis();

  if (ledMode != LED_MODE_STANDBY && (long)(now - ledModeUntilMs) >= 0) {
    ledMode = LED_MODE_STANDBY;
  }

  // ===== Langsames sinusförmiges Fade im Standby =====
  if (ledMode == LED_MODE_STANDBY) {
    if (now - lastLedUpdateMs >= 25) {
      lastLedUpdateMs = now;

      static float phase = 0.0f;

      // kleiner = langsamer
      phase += 0.02f;
```

```
      if (phase > 2.0f * PI) {
        phase -= 2.0f * PI;
      }

      int brightness = (int)((sin(phase) * 0.5f + 0.5f) * 255.0f);

      ledcWrite(ledTuerPin, brightness);
      ledcWrite(ledTorPin, brightness);
    }
    return;
  }

  // ===== Event-Blinken =====
  if (now - lastBlinkToggleMs >= 100) {
    lastBlinkToggleMs = now;
    ledBlinkState = !ledBlinkState;
  }

  if (ledMode == LED_MODE_SHOW_TUER) {
    ledcWrite(ledTuerPin, ledBlinkState ? 255 : 0);
    ledcWrite(ledTorPin, 255);
    return;
  }

  if (ledMode == LED_MODE_SHOW_TOR) {
    ledcWrite(ledTuerPin, 255);
    ledcWrite(ledTorPin, ledBlinkState ? 255 : 0);
    return;
  }
}

bool publishStatus() {
  if (!connectWiFi()) return false;
  if (!ensureMqttConnected()) return false;

  String json = buildDebugJson();
  bool ok = mqttClient.publish(MQTT_TOPIC_STATUS, json.c_str(), true);

  if (ok) {
    Serial.println("MQTT Status (JSON) gesendet");
  } else {
    Serial.println("MQTT Status publish fehlgeschlagen");
  }

  mqttClient.loop();
  return ok;
}

// =====================================================
// Pending Flags
// =====================================================
```

```
bool isMqttPendingTuer() {
  preferences.begin("klingel", true);
  bool pending = preferences.getBool("mqttTuer", false);
  preferences.end();
  return pending;
}

bool isMqttPendingTor() {
  preferences.begin("klingel", true);
  bool pending = preferences.getBool("mqttTor", false);
  preferences.end();
  return pending;
}

void clearMqttPendingTuer() {
  preferences.begin("klingel", false);
  preferences.putBool("mqttTuer", false);
  preferences.end();
}

void clearMqttPendingTor() {
  preferences.begin("klingel", false);
  preferences.putBool("mqttTor", false);
  preferences.end();
}

void markMqttPendingTuer() {
  preferences.begin("klingel", false);
  preferences.putBool("mqttTuer", true);
  preferences.end();
}

void markMqttPendingTor() {
  preferences.begin("klingel", false);
  preferences.putBool("mqttTor", true);
  preferences.end();
}

void processPendingMqtt() {
  if (isMqttPendingTuer()) {
    if (publishUnixTimeToTopic(MQTT_TOPIC_TUER)) {
      clearMqttPendingTuer();
    }
  }

  if (isMqttPendingTor()) {
    if (publishUnixTimeToTopic(MQTT_TOPIC_TOR)) {
      clearMqttPendingTor();
    }
  }
}
```

```
// ========================================================
// Web / Debug / OTA
// ========================================================
String buildDebugJson() {
  String json;
  json.reserve(1700);

  json += "{";
  json += "\"wifi\":\"";
  json += (WiFi.status() == WL_CONNECTED ? "connected" : "disconnected");
  json += "\",";
  json += "\"ip\":\"";
  json += WiFi.localIP().toString();
  json += "\",";
  json += "\"rssi\":";
  json += WiFi.RSSI();
  json += ",";
  json += "\"uptime\":";
  json += millis() / 1000UL;
  json += ",";
  json += "\"free_heap\":";
  json += ESP.getFreeHeap();
  json += ",";
  json += "\"state\":\"";
  json += lastState;
  json += "\",";
  json += "\"source\":\"";
  json += lastSource;
  json += "\",";
  json += "\"adcTorValue\":";
  json += adcTorValue;
  json += ",";
  json += "\"adcTorThreshold\":";
  json += adcTorThreshold;
  json += ",";
  json += "\"lastTorState\":";
  json += (lastTorState ? "true" : "false");
  json += ",";
  json += "\"lastDecodedLen\":";
  json += lastDecodedLen;
  json += ",";
  json += "\"lastDecodedHex\":\"";
  json += lastDecodedHex;
  json += "\",";
  json += "\"mqttPendingTuer\":";
  json += (isMqttPendingTuer() ? "true" : "false");
  json += ",";
  json += "\"mqttPendingTor\":";
  json += (isMqttPendingTor() ? "true" : "false");
  json += ",";
  json += "\"bellPulseActive\":";
```

```
  json += (bellPulseActive ? "true" : "false");
  json += ",";
  json += "\"hostname\":\"";
  json += OTA_HOSTNAME;
  json += "\",";
  json += "\"mac\":\"";
  json += WiFi.macAddress();
  json += "\"";
  json += "}";

  return json;
}

void setupHttpRoutes() {
  httpServer.on("/", HTTP_GET, []() {
    String html;
    html.reserve(3200);

    html += "<!doctype html><html><head><meta charset='utf-8'>";
    html += "<meta name='viewport' content='width=device-width,initial-
scale=1'>";
    html += "<title>HSHB Klingel</title></head><body>";
    html += "<h1>HSHB Klingel</h1>";
    html += "<p><b>IP:</b> " + WiFi.localIP().toString() + "</p>";
    html += "<p><b>RSSI:</b> " + String(WiFi.RSSI()) + "</p>";
    html += "<p><b>Uptime:</b> " + String(millis() / 1000UL) + " s</p>";
    html += "<p><b>State:</b> " + lastState + "</p>";
    html += "<p><b>Source:</b> " + lastSource + "</p>";
    html += "<p><b>ADC Tor:</b> " + String(adcTorValue) + "</p>";
    html += "<p><b>ADC Threshold:</b> " + String(adcTorThreshold) + "</p>";
    html += "<p><b>Last Tor State:</b> " + String(lastTorState ? "true" :
"false") + "</p>";
    html += "<p><b>Decoded Len:</b> " + String(lastDecodedLen) + "</p>";
    html += "<p><b>Decoded Hex:</b> " + lastDecodedHex + "</p>";
    html += "<p><b>MQTT pending Tür:</b> " + String(isMqttPendingTuer() ?
"true" : "false") + "</p>";
    html += "<p><b>MQTT pending Tor:</b> " + String(isMqttPendingTor() ?
"true" : "false") + "</p>";
    html += "<p><b>Bell Pulse:</b> " + String(bellPulseActive ? "true" :
"false") + "</p>";
    html += "<p><b>MAC:</b> " + WiFi.macAddress() + "</p>";
    html += "<p><a href='/debug'>Debug JSON</a></p>";
    html += "<p><a href='/update'>Firmware Update</a></p>";
    html += "</body></html>";

    httpServer.send(200, "text/html", html);
  });

  httpServer.on("/debug", HTTP_GET, []() {
    httpServer.send(200, "application/json", buildDebugJson());
  });
```

```
  httpUpdater.setup(&httpServer, "/update", OTA_USER, OTA_PASS);
  httpServer.begin();
}

void setupArduinoOta() {
  ArduinoOTA.setHostname(OTA_HOSTNAME);
  ArduinoOTA.setPassword(OTA_PASS);
  ArduinoOTA.begin();
}

// =======================================================
// MQTT Callback
// =======================================================
void mqttCallback(char* topic, byte* payload, unsigned int length) {
  String topicStr = String(topic);
  String payloadStr;

  for (unsigned int i = 0; i < length; i++) {
    payloadStr += (char)payload[i];
  }

  payloadStr.trim();

  Serial.print("MQTT empfangen | topic=");
  Serial.print(topicStr);
  Serial.print(" | payload=");
  Serial.println(payloadStr);

  if (payloadStr != "test") return;

  if (topicStr == MQTT_TOPIC_TUER_TEST) {
    Serial.println("MQTT TEST -> TUER");
    triggerBell(true);
  } else if (topicStr == MQTT_TOPIC_TOR_TEST) {
    Serial.println("MQTT TEST -> TOR");
    triggerTorEvent("mqtt-test");
  }
}

// =======================================================
// Klingeltrigger
// =======================================================
void triggerBell(bool fromButton) {
  unsigned long now = millis();
  if (now - lastBellTriggered < BellDebounceMs) return;

  Serial.println("Bell Triggered!");

  ledMode = LED_MODE_SHOW_TUER;
  ledModeUntilMs = now + LedEventDurationMs;
```

```
  lastBellTriggered = now;

  digitalWrite(bellPin, HIGH);
  bellPulseStartMillis = now;
  bellPulseActive = true;

  lastSource = fromButton ? "button" : "bus";
  lastState  = "BELL";

  markMqttPendingTuer();
  processPendingMqtt();
  publishStatus();
}

void triggerTorEvent(const char* source) {
  unsigned long now = millis();

  if (now - lastTorTriggerMillis < TorDebounceMs) {
    Serial.println("TOR blockiert: 11s Sperre aktiv");
    return;
  }

  lastTorTriggerMillis = now;

  ledMode = LED_MODE_SHOW_TOR;
  ledModeUntilMs = now + LedEventDurationMs;
  lastState = "TOR";
  lastSource = source;

  Serial.print("TOR TRIGGER! source=");
  Serial.println(source);

  markMqttPendingTor();
  processPendingMqtt();
  publishStatus();
}

// =====================================================
// RMT Hilfsfunktionen
// =====================================================
int pulseType(uint16_t us) {
  if (us >= 32  && us <= 52)  return P_ONE;
  if (us >= 103 && us <= 123) return P_ZERO;
  if (us >= 223 && us <= 243) return P_START;
  if (us >= 76  && us <= 96)  return P_STOP;
  return P_BAD;
}

size_t collectLows(const rmt_data_t *symbols, size_t symbolCount, uint16_t
*lows) {
  size_t n = 0;
```

```
  for (size_t i = 0; i < symbolCount && n < maxSyms; ++i) {
    if (symbols[i].level0 && !symbols[i].level1 &&
        symbols[i].duration0 >= 17 && symbols[i].duration0 <= 33 &&
        symbols[i].duration1) {
      lows[n++] = symbols[i].duration1;
    } else if (!symbols[i].level0 && symbols[i].level1 &&
               symbols[i].duration1 >= 17 && symbols[i].duration1 <= 33 &&
               symbols[i].duration0) {
      lows[n++] = symbols[i].duration0;
    } else if (!symbols[i].level0 && pulseType(symbols[i].duration0) !=
P_BAD) {
      lows[n++] = symbols[i].duration0;
    } else if (!symbols[i].level1 && pulseType(symbols[i].duration1) !=
P_BAD) {
      lows[n++] = symbols[i].duration1;
    }
  }

  return n;
}

int decodeByte(const uint16_t *pulses) {
  if (pulseType(pulses[0]) != P_START) return -1;

  uint8_t value = 0;
  for (int i = 0; i < 8; ++i) {
    int bit = pulseType(pulses[i + 1]);
    if (bit < 0 || bit > 1) return -1;
    value |= bit << i;
  }
  return value;
}

bool decodeMessage(const rmt_data_t *symbols, size_t symbolCount, uint8_t
*msg, size_t &len) {
  uint16_t lows[maxSyms];
  size_t lowCount = collectLows(symbols, symbolCount, lows);

  len = 0;
  for (size_t i = 0; i + 9 < lowCount && len < bufferSize;) {
    if (pulseType(lows[i]) != P_START) {
      ++i;
      continue;
    }

    int value = decodeByte(lows + i);
    if (value < 0) {
      ++i;
      continue;
    }
```

```
    msg[len++] = value;
    i += 10;

    if (i < lowCount && lows[i] >= 76) {
      ++i;
    }
  }

  return len > 0;
}

bool messageContains(const uint8_t *msg, size_t len, const uint8_t *what,
size_t whatLen) {
  if (whatLen > len) return false;

  for (size_t i = 0; i + whatLen <= len; ++i) {
    if (!memcmp(msg + i, what, whatLen)) return true;
  }
  return false;
}

// =====================================================
// Setup
// =====================================================
void setup() {
  Serial.begin(115200);
  delay(500);

  ledcAttach(ledTuerPin, ledFreq, ledResolution);
  ledcAttach(ledTorPin, ledFreq, ledResolution);

  ledcWrite(ledTuerPin, 0);
  ledcWrite(ledTorPin, 0);

  pinMode(bellPin, OUTPUT);
  digitalWrite(bellPin, LOW);

  pinMode(testBellPin, INPUT_PULLUP);
  pinMode(adcTorPin, INPUT);

  analogReadResolution(12);
  analogSetPinAttenuation(adcTorPin, ADC_11db);

  WiFi.mode(WIFI_STA);
  esp_wifi_set_mac(WIFI_IF_STA, &newMACAddress[0]);

  mqttClient.setServer(MQTT_HOST, MQTT_PORT);
  mqttClient.setCallback(mqttCallback);
  mqttClient.setBufferSize(2048);
  mqttClient.setKeepAlive(30);
  mqttClient.setSocketTimeout(2);
```

```
  connectWiFi();
  ensureTimeValid();
  ensureMqttConnected();

  if (MDNS.begin(OTA_HOSTNAME)) {
    MDNS.addService("http", "tcp", 80);
  }

  setupHttpRoutes();
  setupArduinoOta();

  rmtInit(busPin, RMT_RX_MODE, RMT_MEM_NUM_BLOCKS_2, 1000000);
  rmtSetRxMinThreshold(busPin, 3);
  rmtSetRxMaxThreshold(busPin, 1000);
  rmtReadAsync(busPin, symbols, &symbolCount);

  publishStatus();
}

// ======================================================
// Loop
// ======================================================
void loop() {
  unsigned long now = millis();

  updateLeds();
  connectWiFi();
  ensureMqttConnected();
  mqttClient.loop();

  httpServer.handleClient();
  ArduinoOTA.handle();

  adcTorValue = analogRead(adcTorPin);

  bool currentTorState = (adcTorValue > adcTorThreshold);

  // nur steigende Flanke
  if (!lastTorState && currentTorState) {
    Serial.print("TOR Flanke erkannt, ADC=");
    Serial.println(adcTorValue);
    triggerTorEvent("adc");
  }

  lastTorState = currentTorState;

  bool currentButtonState = digitalRead(testBellPin);
  if (lastButtonState == HIGH && currentButtonState == LOW &&
      (now - lastButtonTriggerMillis > ButtonDebounceMs)) {
    lastButtonTriggerMillis = now;
    triggerBell(true);
```

```
  }
  lastButtonState = currentButtonState;

  if (rmtReceiveCompleted(busPin)) {
    uint8_t msg[bufferSize];
    size_t len = 0;

    decodeMessage(symbols, symbolCount, msg, len);

    lastDecodedLen = len;
    lastDecodedHex = "";
    for (size_t i = 0; i < len; ++i) {
      char tmp[4];
      snprintf(tmp, sizeof(tmp), "%02X", msg[i]);
      if (i) lastDecodedHex += " ";
      lastDecodedHex += tmp;
    }
    if (lastDecodedHex.length() == 0) lastDecodedHex = "-";

    symbolCount = maxSyms;
    rmtReadAsync(busPin, symbols, &symbolCount);

    if (messageContains(msg, len, hackerspaceBell, sizeof(hackerspaceBell))
||
        messageContains(msg, len, hackerspaceBell2,
sizeof(hackerspaceBell2))) {
      triggerBell(false);
    }
  }

  if (bellPulseActive && (now - bellPulseStartMillis > BellPulseMs)) {
    digitalWrite(bellPin, LOW);
    bellPulseActive = false;
    if (lastState == "BELL") {
      lastState = "IDLE";
    }
  }

  if (now - lastStatusMillis > StatusMs) {
    lastStatusMillis = now;
    publishStatus();
  }

  if (now - lastDebugMillis > Debuginterval) {
    lastDebugMillis = now;
    Serial.print("state=");
    Serial.print(lastState);
    Serial.print(" source=");
    Serial.print(lastSource);
    Serial.print(" adc=");
    Serial.print(adcTorValue);
```

```
    Serial.print(" torState=");
    Serial.println(currentTorState ? "1" : "0");
  }

  delay(1);
}
```

From:
https://wiki.hackerspace-bremen.de/ - **Hackerspace Bremen e.V.**

Permanent link:
https://wiki.hackerspace-bremen.de/sonstiges/tuerklingel

Last update: **2026-05-07 02:21**