Inhaltsverzeichnis

Water Flow Sound Detector (ESP32 + MAX9814)..... 1

1. Goal..... 1

2. Design decisions..... 2

3. Hardware..... 2

4. Firmware..... 3

5. Calibration procedure..... 4

6. Testing on Ubuntu 26.04 with VS Code..... 5

7. Using Claude with PlatformIO..... 5

8. Known limitations and risks..... 6

Wi-Fi Notifications..... 6

1. Overview..... 6

2. Hardware..... 6

3. How it works..... 7

4. Wi-Fi notifications..... 7

5. Troubleshooting log..... 8

6. Current recommended sendMessage()..... 8

7. Calibration..... 9

8. Debugging with the serial monitor..... 9

9. Security notes..... 9

Water Flow Sound Detector (ESP32 + MAX9814)

Project documentation, 6 November 2026. Summary of the design discussion, hardware, firmware, testing and open items.

1. Goal

Detect the sound of running water in a water pipe on an ESP32 and switch an alarm when the water has been running for more than one minute.

2. Design decisions

No machine learning needed. Running water produces sustained broadband noise (a hiss), not a distinct pattern. That makes the task an energy measurement in a frequency band, averaged over time, which is simple, cheap and reliable on an ESP32. Because the water runs for over a minute, long averaging windows filter out short disturbances.

ML only as a fallback. TinyML (TensorFlow Lite for Microcontrollers, Edge Impulse) would only be worth it if false alarms from other sources cannot be solved with simpler means.

Detection method.

1. Read audio in short blocks (512 samples).
2. Compute the energy at several frequencies with the Goertzel algorithm (a cheap alternative to a full FFT), after removing the DC offset and applying a Hann window.
3. Smooth the level with an exponential moving average, in dB.
4. Compare to two thresholds with hysteresis: a higher one to switch to "flowing", a lower one to switch back.
5. Start a timer when flow is detected. Trigger the alarm if it lasts longer than 60 seconds. Reset only if the level stays low for 5 seconds.

Microphone placement. An air microphone in the room also picks up TV, speech and appliances. Fixing the microphone directly against the pipe (tape or clamp, with foam) gives a strong signal and much less room-noise sensitivity.

Microphone choice. The first version used an INMP441 (digital I2S MEMS microphone). The current version uses a **MAX9814**, an analog microphone amplifier with automatic gain control (AGC), read through the ESP32's ADC.

3. Hardware

Part	Purpose
ESP32 (classic DevKit assumed; ESP32-S3 also possible, pin numbers differ)	Controller
MAX9814 microphone module	Sound input (analog)
Active buzzer or LED with resistor	Alarm output

Wiring (current version)

MAX9814 / part	ESP32
VDD	3.3 V (not 5 V, the output must stay below 3.3 V)
GND	GND
OUT	GPIO 35 (ADC1, input-only)

MAX9814 / part	ESP32
GAIN	VDD (40 dB, lowest maximum gain)
AR	GND (slowest AGC release)
Buzzer / LED	GPIO 4
Onboard LED (flow indicator)	GPIO 2

The GAIN and AR settings can differ between breakout boards, so check the datasheet of the board in use. Keep wires short. A 100 nF capacitor across the supply close to the module helps reduce ADC noise.

Pin notes

- **Microphone input:** must be an ADC1 pin, because ADC2 does not work while Wi-Fi is active. On the classic ESP32 the options are GPIO 32, 33, 34, 35, 36 and 39. GPIO 34, 35, 36 and 39 are input-only. Attention: on some boards Pin 35, 36 and 37 are used for PSRAM
- **Outputs (alarm, LED):** any normal output GPIO works. Good choices are 16, 17, 18, 19, 21, 22, 23 (and 13, 14, 25, 26, 27). Avoid GPIO 6 to 11 (flash), 34/35/36/39 (input-only) and the boot-strapping pins 0, 2, 5, 12, 15 where a load could interfere with booting.
- **ESP32-S3 / S2:** ADC1 is GPIO 1 to 10. Avoid strapping pins (0, 3, 45, 46), USB pins (19, 20), and the flash/PSRAM pins (26 to 32, some modules also 33 to 37).
- On some boards (for example TTGO, Heltec) GPIO 35 is also connected to a battery-voltage divider.
- A GPIO can only supply about 20 to 40 mA. A relay or a loud buzzer needs a transistor or MOSFET.

4. Firmware

Files

- `water_flow_detector.ino`: first version for the INMP441 (I2S microphone: SCK GPIO 26, WS GPIO 25, SD GPIO 33, L/R to GND).
- `water_flow_detector_max9814.ino`: current version for the MAX9814 on GPIO 35.

Both are Arduino-framework sketches. The MAX9814 version works on Arduino-ESP32 2.x and 3.x.

How the MAX9814 version works

- Samples the ADC at 10 kHz (software-paced with `micros()`; small timing jitter is acceptable because only broadband noise energy is measured). The analysed band is therefore 0.5 to 4.5 kHz, with Goertzel at nine frequencies from 500 Hz to 4500 Hz.
- Removes the DC offset (the MAX9814 output sits at about 1.25 V), scales to about -1..1 and applies a Hann window.
- Sums the Goertzel powers, converts to dB and smooths with an exponential average.
- Runs the hysteresis and timer logic described in section 2.

- Beeps the alarm output (300 ms on, 300 ms off) and stops after the maximum alarm time or when the water stops.

Parameters

Parameter	Default	Meaning
`CALIBRATION_MODE`	`true`	Only print levels, no alarm
`MIC_PIN`	35	MAX9814 output (ADC1)
`ALARM_PIN`	4	Buzzer / relay driver
`LED_PIN`	2	Shows "flow detected"
`SAMPLE_RATE`	10000 Hz	ADC sampling rate
`BLOCK_SIZE`	512	Samples per analysis block (about 51 ms)
`LEVEL_HIGH`	-40 dB	Above this, water is considered flowing (placeholder)
`LEVEL_LOW`	-46 dB	Below this, water is considered stopped (placeholder)
`SMOOTHING`	0.97	Smoothing factor, about 1.5 s time constant
`FLOW_ALARM_MS`	60000	Flow duration before the alarm
`STOP_DEBOUNCE_MS`	5000	Low-level time needed to reset
`ALARM_MAX_MS`	120000	Maximum beeping duration

The dB thresholds are placeholders and must be set from real measurements.

5. Calibration procedure

1. Leave `CALIBRATION_MODE` set to `true`, upload, and open the serial monitor at 115200 baud.
2. Fix the microphone to the pipe.
3. Watch the smoothed dB value for about 20 seconds in each situation: water off in a quiet room, water running, talking or TV nearby, toilet flush or washing machine if available.
4. Check the `p2p` value (should be well above 0 but far below 4095) and `rate` (should be close to 10000 Hz). If it prints `CLIPPING!`, lower the gain or move the microphone slightly away from the pipe.
5. Set `LEVEL_HIGH` and `LEVEL_LOW` between the "off" and "running" values, with `LEVEL_LOW` a few dB below `LEVEL_HIGH`.
6. Set `CALIBRATION_MODE` to `false`, upload again, and test with a short `FLOW_ALARM_MS` (for example 10000). Set the final 60000 afterwards.

Decision rule. If "on" and "off" are at least about 6 dB apart, the current detection is sufficient. If they are too close together, the AGC is flattening the signal, and the detection should be changed to compare the ratio between frequency bands, which does not depend on overall volume.

6. Testing on Ubuntu 26.04 with VS Code

Option 1: real ESP32 with PlatformIO (recommended)

```
sudo apt install python3-venv python3-pip git
sudo usermod -aG dialout $USER     # serial port permission, log out and in afterwards
sudo apt remove brltty             # often grabs CH340/CP210x USB-serial chips
```

Install the PlatformIO IDE extension in VS Code, create a project (board *Espressif ESP32 Dev Module*, framework *Arduino*), copy the sketch into `src/main.cpp`, and use:

```
[env:esp32dev]
platform = espressif32
board = esp32dev
framework = arduino
monitor_speed = 115200
```

Use the Build, Upload and Serial Monitor buttons in the bottom status bar (or `pio run`, `pio run -t upload`, `pio device monitor`). PlatformIO was not verified against Ubuntu 26.04 specifically. If the extension complains about the Python version, install a slightly older Python via `uv` or `pyenv`, or use the PlatformIO CLI (`pipx install platformio`).

Option 2: simulation

The Wokwi for VS Code extension can simulate an ESP32 and is useful for checking logic, serial output and timers. It is uncertain how well it simulates a microphone input.

Option 3: signal-processing test on the PC

The Goertzel level calculation is plain C++ and can be tested natively with `g++ -O2 test.cpp -o test && ./test`, using synthetic signals (quiet room, 200 Hz hum, white-noise "water-like" hiss). The hiss should come out well above the other two. Absolute values will differ from the real microphone, so calibration on the pipe is still required.

7. Using Claude with PlatformIO

PlatformIO runs inside VS Code, so the Claude Code extension can be used in the same editor and project folder.

1. In the VS Code Extensions view, search for "Claude Code" and check that the publisher is Anthropic. It requires VS Code 1.98.0 or newer and a Claude login or API key.
2. Open the PlatformIO project and a source file such as `src/main.cpp`. The Spark icon in the top-right of the editor toolbar opens Claude Code.
3. Sign in with a Claude account when prompted.

Typical uses: asking for code changes (shown as a diff for approval), running `pio run` and fixing compiler errors, and choosing thresholds from pasted serial output. The standalone CLI is only needed for running `claude` in the integrated terminal. Claude cannot see the hardware, only the build output and text that

is provided. Which plans include Claude Code can change, so check https://code.claude.com/docs for current requirements.

8. Known limitations and risks

- **AGC of the MAX9814** cannot be switched off. It raises the gain in quiet conditions and lowers it for loud sounds, which reduces the difference between "off" and "on". This has not been tested on a real pipe.
- **Software-paced ADC sampling** has some timing jitter, and the ESP32 ADC is noisy and non-linear. Both are acceptable for a broadband energy measurement but not for precise audio work.
- **No anti-aliasing filter**: energy above 5 kHz can fold into the analysed band at the 10 kHz sampling rate.
- **False alarms** are possible from pumps, heating circulation, washing machines and dishwashers, which also make pipe noise for minutes. Comparing several frequency bands (their ratio) is the next step if this becomes a problem.
- **Thresholds** are placeholders until calibrated on the real pipe.
- If the goal is leak or flow detection rather than sound specifically, a clamp-on flow sensor or an in-line flow meter is more reliable.

Wi-Fi Notifications

1. How it works

- **Acquisition:** `acquireBlock()` reads 512 samples at 10 kHz from the ADC (about 51 ms per block).
- **Analysis:** `computeBandLevelDb()` removes the DC offset, applies a Hann window and computes the power at 9 frequencies (500 to 4500 Hz) with the Goertzel algorithm. The sum is converted to dB.
- **Smoothing:** exponential smoothing in the dB domain (`SMOOTHING` = 0.97, about 1.5 s).
- **Detection:** `updateDetection()` is a small state machine with hysteresis. Flow starts above `LEVEL_HIGH`. It ends when the level stays below `LEVEL_LOW` for `STOP_DEBOUNCE_MS`. The alarm fires after `FLOW_ALARM_MS` of continuous flow.
- **Alarm output:** `updateAlarmOutput()` beeps 300 ms on / 300 ms off for at most `ALARM_MAX_MS`.
- **Networking:** a separate FreeRTOS task sends the messages (see section 4).

2. Wi-Fi notifications

Design

The sampling loop must never wait for the network. An HTTP request can block for seconds and would corrupt the audio blocks and the smoothing. Therefore:

- The main loop only calls `notify(MSG_FLOWING)` or `notify(MSG_ALARM)`. This puts a message into a FreeRTOS queue and returns immediately.
- `networkTask()` runs in the background, connects to Wi-Fi, waits for messages in the queue and sends them with an HTTP POST (up to 3 attempts).

Service

The example uses ntfy.sh. Install the ntfy app and subscribe to your topic. Anyone who knows the topic name can read the messages, so choose a long, random name. Other services (Telegram, MQTT, Pushover) only require changes in `sendMessage()`.

Settings

```
const char* WIFI_SSID  = "your-ssid";
const char* WIFI_PASS  = "your-password";
const char* NOTIFY_URL = "https://ntfy.sh/your-random-topic";
```

3. Troubleshooting log

Problem 1: No message was sent

Cause: The two `notify(...)` calls were missing in `updateDetection()`. The queue and task existed, but nothing wrote into the queue.

Fix: Add `notify(MSG_FLOWING);` where flow is detected and `notify(MSG_ALARM);` where the alarm is activated.

Problem 2: Crash after sending (`HTTP -11`, then Guru Meditation `InstrFetchProhibited`, `PC: 0x00000000`)

Causes:

- In `sendMessage()`, the `HTTPClient` was declared before the network clients. It was therefore destroyed after them and accessed an already destroyed client, which caused a call through a null pointer.
- ntfy redirects `http://` to `https://`. Over plain HTTP this ended in a read timeout (`-11`).

Fixes:

6. Declare the client first, then `HTTPClient`, so the client is destroyed last.
7. Use only `WiFiClientSecure` with `setInsecure()` and an `https://` URL.
8. Increase the task stack from 8192 to 12288 bytes, because TLS needs more memory.

Observation: "Alarm" appeared before "Water is flowing" in the log

The network task first has to connect to Wi-Fi, which takes several seconds. The first message is therefore delayed. To shorten this, call `WiFi.begin()` in `setup()` instead of inside the task.

4. Current recommended `sendMessage()`

```
static bool sendMessage(const char* text) {
  WiFiClientSecure client;        // declared first, destroyed last
  client.setInsecure();
  client.setTimeout(10);          // seconds

  HTTPClient http;
  if (!http.begin(client, NOTIFY_URL)) return false;

  http.setTimeout(10000);
  http.addHeader("Content-Type", "text/plain");
  http.addHeader("Title", "Water monitor");
  int code = http.POST((uint8_t*)text, strlen(text));
  http.end();

  Serial.printf("[net] \"%s\" -> HTTP %d\n", text, code);
  return code >= 200 && code < 300;
}
```

Task creation in `setup()`:

```
if (!CALIBRATION_MODE) {
  msgQueue = xQueueCreate(8, sizeof(MsgType));
  xTaskCreate(networkTask, "net", 12288, nullptr, 1, nullptr);
}
```

5. Calibration

1. Set `CALIBRATION_MODE` to `true` and open the serial monitor at 115200 baud.
2. Note the dB values with water off and on.
3. Set `LEVEL_HIGH` and `LEVEL_LOW` between these values (about 2 dB apart for hysteresis).
4. Set `CALIBRATION_MODE` back to `false`.

Wi-Fi activity can slightly shift ADC readings, so check the thresholds again with Wi-Fi running. The serial output also shows clipping and warns if the ADC is too slow for the chosen sample rate.

7. Debugging with the serial monitor

Output	Meaning
`[net] "Alarm" -> HTTP 200`	Message sent successfully
`[net] ... -> HTTP -1`	Connection problem (Wi-Fi or DNS)
`[net] ... -> HTTP -11`	Read timeout
No `[net]` line	Nothing was put into the queue, check the `notify()` calls

8. Security notes

- Do not share code containing your Wi-Fi password. If it was posted publicly, change it.
- `setInsecure()` disables certificate checks. This is acceptable for simple notifications, but do not send sensitive data this way.
- Use a long random ntfy topic, since topics are public by default.