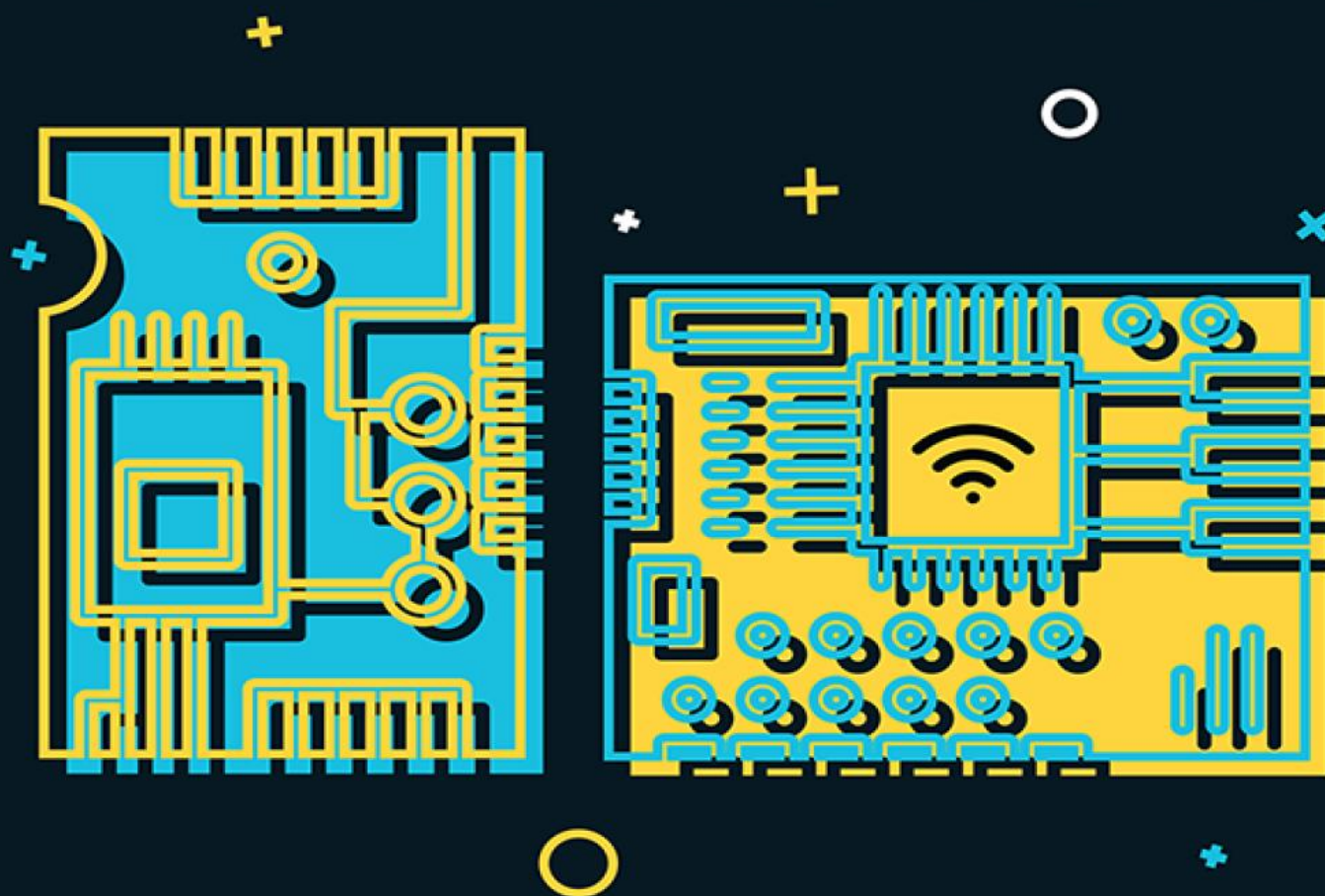


Andrzej „SunRiver” Gromczyński

W LABIRYNCIE IoT

Budowanie urządzeń z wykorzystaniem układów

ESP8266 i ESP32



Ten ebook jest chroniony znakiem wodnym

ebookpoint.pl Kopia dla:

Detonator Damian
blow@it.up

Andrzej Gromczyński

W labiryncie IoT

Budowanie urządzeń z wykorzystaniem
układów ESP8266 i ESP32

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Tomasz Gojowy

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą AdobeStock.

HELION S.A.

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

https://helion.pl/user/opinie/wlablo_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-289-2273-0

Copyright © Helion S.A. 2025

- [Kup w wersji papierowej](#)
- [Poleć książkę na Facebook.com](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Drodzy Czytelnicy,

Chciałbym serdecznie podziękować za towarzyszenie mi w tej niezwykłej podróży przez labirynt IoT z ESP8266 i ESP32. Wasza ciekawość i gotowość do odkrywania czynią tę książkę wyjątkową. Bez Waszego entuzjazmu i pasji ta podróż nie byłaby tak inspirująca.

Chciałbym też podziękować wszystkim przyjaciołom z naszego małego forum dostępnego pod adresem <https://forum.lothar-team.pl>, którzy wspierali mnie na etapie powstawania tej książki, i choć ukrywałem ją przed nimi, i tak się domyślali, nieświadomie dawali wskazówki i rady oraz jakże nieocenioną pomoc, którą jest wsparcie. Dodatkowo kolega Xbary wspomagał mnie zdjęciami modułów, których akurat nie miałem pod ręką, a które znalazły się w jego zbiorach.

Teraz gdy macie klucz do labiryntu, zapraszam Was do aktywnego uczestnictwa w tej przygodzie. Niech każdy rozdział będzie dla Was wyzwaniem do eksperymentowania, do przekraczania własnych granic i do tworzenia czegoś wyjątkowego. Wasze pytania, uwagi i pomysły stanowią integralną część tej książki.

Dziękuję, że jesteście ze mną, i mam nadzieję, że każda strona tej książki przyniesie Wam inspirację i nowe umiejętności. Niech ta podróż stanie się dla Was niezapomnianym doświadczeniem, co otworzy przed Wami nieograniczone możliwości.

Gotowi na niezapomniane chwile w świecie Internetu Rzeczy? Czas zanurzyć się w książce *W labiryncie IoT. Budowanie urządzeń z wykorzystaniem układów ESP8266 i ESP32*. Czekają na Was fascynujące eksperymenty, kreatywne wyzwania i niekończące się inspiracje.

Z przyjemnością zapraszam Was do tego niezwykłego świata!

Z poważaniem,
SunRiver

Wstęp

Witaj w fascynującym świecie Internetu Rzeczy (IoT), gdzie możliwości są tak bezgraniczne jak Twoja wyobraźnia. Ta książka, *W labiryncie IoT. Budowanie urządzeń z wykorzystaniem układów ESP8266 i ESP32*, zaprasza Cię do pełnej kreatywności podróży przez labirynt możliwości, jakie oferują układy ESP8266 i ESP32.

Nie jest tajemnicą, że współczesne rozwiązania IoT stanowią klucz do zrozumienia i kształtowania przyszłości. W tym kontekście układy ESP8266 i ESP32 wyróżniają się swoją wszechstronnością, niskim kosztem i potężną mocą obliczeniową, co czyni je idealnym narzędziem dla entuzjastów, hobbystów i profesjonalistów pragnących przeforować swoje pomysły w realnym świecie.

Czego możesz się spodziewać

Ta książka nie jest zwykłym przewodnikiem po układach ESP; to podróż przez praktyczne zastosowania tych mikrokontrolerów w amatorskich projektach IoT. Poprzez serię inspirujących projektów krok po kroku wprowadzę Cię w świat programowania, interfejsu z urządzeniami, zbierania danych i tworzenia inteligentnych urządzeń.

Rozpocznę od podstaw, aby uczynić tę książkę dostępną dla wszystkich, bez względu na poziom doświadczenia. Ci, którzy są już zaznajomieni z podstawami, znajdą tu wyzwania do podjęcia, otwierające drzwi do zaawansowanych technik i strategii projektowania.

Dlaczego ESP8266 i ESP32?

To pytanie jest jak fundament, na którym zbudowano tę książkę. ESP8266 i ESP32 to nie tylko mikrokontrolery; to narzędzia transformujące pomysły w rzeczywistość. Ich wszechstronność pozwala na stworzenie niemal każdego rodzaju urządzenia IoT, od prostych czujników po zaawansowane systemy monitoringu.

Dla kogo jest ta książka?

Jeśli jesteś hobbystą, studentem, programistą czy inżynierem, który pragnie zagłębić się w świat IoT i wykorzystać potencjał układów ESP8266 i ESP32, to ta książka jest dla Ciebie. Niezależnie od tego, czy dopiero zaczynasz swoją przygodę z IoT, czy jesteś doświadczonym twórcą, znajdziesz tu inspirację do realizacji swoich projektów.

Przygotuj się na niezapomnianą podróż przez labirynt IoT, gdzie amatorskie projekty przeradzają się w profesjonalne wyniki. Czas zanurzyć się w świecie ESP8266 i ESP32!

CZĘŚĆ I. Podstawy świata IoT

Zanim zagłębimy się w skomplikowane projekty, zaczniemy od solidnych podstaw. W pierwszym rozdziale odkryjecie tajniki programowania układów ESP8266 i ESP32. Nowicjusom zapewnimy jasne i przystępne wyjaśnienia, a dla doświadczonych programistów mamy ciekawe wyzwania.

W tym miejscu chciałbym też od razu zaznaczyć, że nie będę pisał o języku C, jak robi wielu autorów książek o mikrokontrolerach. Tak więc przykro mi, jeśli się zawiedliście, bo liczyliście na kolejny kurs języka C od podstaw. Nie, nie będę Was na siłę uczył programowania, ponieważ zwyczajnie tego nie umiem, a nie brakuje naprawdę dobrych książek o C i dobrych nauczycieli. Jednakże mogę Wam obiecać, że będzie tyle C, ile w C go potrzeba. Poznacie też trochę teorii i przykładowego użycia w roli brokera systemów SBC, podobnych do popularnego Raspberry Pi, i trochę poruszania się po konsoli w Linuxie. Będę się starał pokazać Wam w pigułce różne podejścia do tego samego tematu, co być może zainspiruje Was do poszukiwania własnych rozwiązań.

W tym rozdziale, jak to na początku, przedstawię Wam możliwości układów 8266 i ESP32 oraz różnice między nimi. Choć ten pierwszy jest powoli wypierany przez następców, wciąż ma ogromny potencjał.

1. Trochę historii czy wprowadzenie?

Właściwie to trochę sam nie jestem zdecydowany, jak miałbym zacząć. Ale zakładam, że trzymanie się ścisłych faktów i mitów jest bezcelowe. Zaczniemy od początku, ale nic pewnego. Was tu nie czeka, nie jest to bowiem książka historyczna, więc nie musimy się sztywno niczego trzymać...

Zatem do dzieła.

Dawno, dawno temu, w tajnym laboratorium ukrytym pod górą, grupa szalonych inżynierów postanowiła stworzyć coś, co zmieniłoby świat elektroniki. Siedząc przy stole z kubkiem kawy w jednej ręce i lutownicą w drugiej, wpadli na genialny pomysł. Postanowili połączyć siły ducha DIY (ang. *do it yourself*, zrób to sam) z magią bezprzewodowego Internetu Rzeczy. ESP8266 narodził się jako malutki czarodziej, który miał moc łączenia urządzeń z internetem, bez żadnych zbędnych ceregieli. Inżynierowie dali mu imię "ESP" — czyli "Elektroniczny Szalony Pomysł". Wydaje się, że jeden z nich musiał nadużywać kofeiny, ponieważ ESP8266 był tak mały, że można go było prawie stracić w elektronicznym śmietniku. Jednak świat chciał jeszcze więcej magii, więc inżynierowie postanowili stworzyć potężniejszą wersję. Tak narodził się ESP32, starszy brat ESP8266. Był bardziej zaawansowany, bardziej potężny i trochę bardziej złożony. Inżynierów, którzy stworzyli ESP32, ogłoszono "Mistrzami Magii Elektronicznej", a także przyznano im order z ziemniaka oraz dodatkowe racje ryżu.

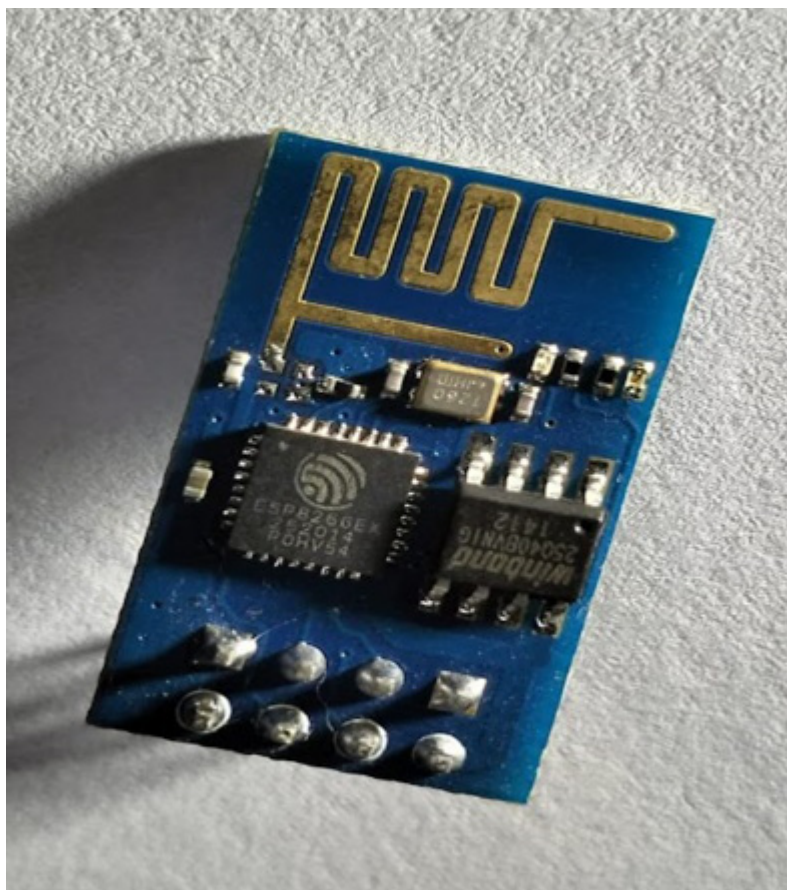
Odtąd ESP8266 i ESP32 stały się nieodłącznymi towarzyszami hobbystów, inżynierów i każdego, kto chciał wpleść trochę magii w swoje projekty. I tak kończy się historia tajemniczego laboratorium, gdzie kawa, kreatywność i trochę szaleństwa doprowadziły do powstania dwóch niesamowitych czarodziejów — ESP8266 i ESP32. Gdy ESP8266 i ESP32 wyruszyły w świat, zaczęły zdobywać serca miłośników techniki. Ludzie zaczęli zaklinać elektroniczne duchy, rzucać czary kodu i łączyć swoje projekty z kosmiczną magią Internetu Rzeczy. ESP8266, nazywany czasem "Młodszym Czarodziejem", był mistrzem prostoty. Wielu z jego zwolenników mówi, że gdy połączysz go z baterią, dwoma drutami i

odrobiną kodu, może zacząć pieścić twoje urządzenia za pomocą fal radiowych. To był prawdziwy czar elektroniki na wyciągnięcie ręki!

ESP32 z kolei zjawił się jako "Starszy Czarodziej". Znany był ze swojej zdolności do obsługi większej liczby zaklęć jednocześnie. Ludzie używali go do tworzenia zaawansowanych projekcików, takich jak inteligentne domy, roboty czy nawet magiczne lusterka, które mówią, jak świetnie wyglądasz każdego ranka. Ostatecznie ESP8266 i ESP32 stały się jak dwie szalone wróżki, które pomagały ludziom spełniać ich elektroniczne marzenia. I chociaż nadal krążą opowiadane szeptem legendy o tym tajemnym laboratorium, jedno jest pewne: dzięki ESP8266 i ESP32 świat stał się bardziej kolorowy i pełen wspaniałych wynalazków przepełnionych magią!

Zapewne tak by mogło całe owo wydarzenie wyglądać, gdyż niejako w każdej bajce jest ziarenko prawdy. Ale tak żartując, okazuje się, że to, co napisałem, ma wiele sensu, ponieważ rok 2014 był bardzo obiecujący dla wielu amatorów, wynalazców wręcz marzących o podłączeniu do sieci internet swoich pomysłów. Oczywiście rozwiązań jest wiele, od połączeń kablowych po interfejsy bezprzewodowe, powszechnie zwane WiFi. Rozwiązania te, szczerze mówiąc, pozostawiają wiele do życzenia, są drogie i kłopotliwe w użyciu. No może nie do końca kłopotliwe, w każdym razie konfiguracja, uciążliwa za sprawą poleceń AT i ze względu na konieczność używania UART-u do komunikacji, nie była, powiedzmy, specjalnie przyjemna i wygodna, już nie mówiąc o szybkości... I nagle z Nienacka (ok. 23 km na północ od Poznania) pojawia się młoda firma wprowadzająca rewolucyjny układ 8266, nie tylko umożliwia konfigurację za pomocą poleceń AT i komunikację przez UART, ale też udostępnia kilka GPIO i możliwość pracy w kilku trybach WiFi.

ESP-01 (rysunek 1.1) jest pierwszym modulem z nowym układem ESP8266EX. Wprawdzie ma niewielki rozmiar, jak i niewiele wyprowadzonych pinów, ale szybko pokazuje swój potencjał.



Rysunek 1.1. Wygląd modułu ESP-01¹

Układ przedstawia się ciekawie, jest tani i jak się szybko okazuje, nie musi być tylko mostem WiFi-UART; jest też całkiem porządnym mikrokontrolerem o wielu możliwościach i dużej wydajności jak na takie tanie maleństwo. Mała firma Espressif stworzyła coś niesamowitego.

Sam pamiętam, jak się pojawił, jak wielu się naśmiewało z chińskiego produktu, jak z popularnego Aliexpress

sprowadzałem pierwsze sztuki modułów ESP01, a później pomniejsze sklepiki oferowały coraz ciekawsze i bogatsze moduły. Ba, na jednym forum wręcz mnie wyklęto za sianie herezji i zamętu, bo opisywałem tak trywialne chińskie wynalazki, które nawet nie miały działać. Tymczasem układy ESP oferowały wiele znanych peryferiów. Wydajny i szybki rdzeń Tensilic Diamond, 32-bitowy RISC taktowany z częstotliwością od 80 do 160 MHz, zawierał ponadto 32 KiB pamięci RAM, 32 KiB pamięci podręcznej instrukcji, 80 KiB pamięci RAM na dane użytkownika i 16 KiB pamięci RAM na dane systemowe ETS. Wbudowane radio WiFi pracowało w standardzie IEEE 802.11b/g/n, a ponadto miało zintegrowany przełącznik TR, Balun, LNA, a nawet wzmacniacz mocy. Pozwalało też na uwierzytelnienie WEP lub WPA/WPA2 i pracę w sieci otwartej. Układ udostępniał ponadto 17 pinów GPIO, których w praktyce było mniej lub więcej, zależnie od modułu, oraz znane peryferia, takie jak SPI, i2s UART na osobnych pinach czy 10-bitowy ADC. Kolejną ciekawostką było to, że układ jako taki nie zawierał wbudowanej pamięci flash. Ta była dołączana w zewnętrznym układzie QSPI, obsługującym maksymalnie 16 MB, choć na modułach zwyczajowo dostępne było od 512 kB do 4 MB. Niebawem też pojawił się układ ESP8285, który miał wbudowane 1 MB pamięci flash i pozwalał na konstruowanie miniaturowych jednoukładowych urządzeń zdolnych do łączenia się z WiFi.

Pojawiło się też oficjalne SDK pozwalające programować układ w języku C oraz porty społecznościowe umożliwiające programowanie modułów w środowisku Arduino. Gdy się okazało, że ten niewielki układzik jest bardzo przystępny, wielu amatorów zaczęło używać go nie tylko jako UART-WiFi, ale też jako alternatywę dla innych układów opartych na mikrokontrolerach i WiFi. Zdecydowanie niższy koszt urządzeń i większa dostępność WiFi w amatorskich rozwiązaniach miały niebagatelny wpływ na rozwój

dziedziny IoT oraz wszelkich dostępnych dziś rozwiązań *smart home*. Rzecz jasna układy ESP8266 z czasem zostały zastąpione wszechobecnymi układami ESP32, które zaoferowały dwa rdzenie taktowane z częstotliwością 240 MHz oraz obsługę WiFi i Bluetooth. Oczywiście nigdy nie jest tak, że od początku wszystko jest super. Widząc sukces układów ESP, wiele firm chciało uszczknąć dla siebie kawałek tortu. W ten sposób powstało wiele rozwiązań różnych producentów, choćby skądinąd ciekawe układy Ameba, produkowane przez firmę Realtek. Ale nie przyjęły się specjalnie, choć użyto w nich rdzenia Cortex firmy ARM, więc bardziej przejrzystego i znanego, a na pewno przyjemniejszego w użytkowaniu od jakiegoś dziwnego Tensilica. Jednak tak to już jest, i to od dawna, w świecie elektroniki. Gdy Arduino święciło pierwsze sukcesy, natychmiast pojawiło się wiele podróbek i podobnych płytek; gdy powstało Raspberry Pi, pomimo wielu ułomności niemal natychmiast wywołało lawinę podobnych rozwiązań. Czasem lepsze, czasem gorsze, tańsze i droższe — na wybór niewątpliwie nie można było narzekać. Sam np. bardzo polubiłem niepozorne układy RP2040... ale to już całkiem inna opowieść.

¹ Źródło: <https://forum.lothar-team.pl>; dostęp 15.06.2024.

2. Sprzętowoisko

— czyli z czym po drodze

Wbrew temu, co myślicie, dobór sprzętu ma tu znaczenie, choć zdecydowanie mniejsze, niż się Wam teraz wydaje. Oczywiście oprzemy się na kilku rozwiązaniach sprzętowych dostępnych w handlu, od poważnych zestawów developerskich sprzedawanych przez producenta "krzemu" po tanie i łatwo dostępne płytki, które zapewne wielu z Was ma już w swojej szufladzie. Niemniej nie ma większego znaczenia to, na czym ja się będę opierał, przedstawiając dane zagadnienie. W każdym razie poczuwam się do obowiązku przedstawienia Wam kilku płytek, które, mam nadzieję, pomogą mi poprowadzić Was przez labirynt.

1. ESP-03 — najmniejszy moduł z ESP8266 i ceramiczną anteną, udostępniający siedem GPIO i jedno ADC. To jego wybrałem na podstawę swoich modułów. Dzięki niemu pokażę Wam, jak łatwo samodzielnie wykonać małe wielkie urządzenie do sterowania.
2. D1 mini — mała płytka w niskiej cenie z modułem ESP-12F i USB do programowania.
3. ESP32 One — tę wyposażoną w ESP32 płytkę lubię ze względu na format popularnego Raspberry Pi Zero, co jest dosyć wygodne podczas projektowania i prototypowania złożonych projektów.
4. SC01 Plus — to zaawansowany twór wyposażony w ESP32-S3 i dotykowy ekran IPS 3,5". Postaram się na jego przykładzie przedstawić Wam zalety tworzenia interfejsów z użyciem różnych narzędzi.

5. ESP-PROG — pojawi się parę razy w książce i zapewne uratuje nas z opresji, gdy utkniemy. I choć programator/debuger nie będzie nam tak bardzo potrzebny, sama jego obecność będzie powodować poczucie bezpieczeństwa.
6. ESP32 jakies tam... Czyli bliżej nieokreślona płytką, ale z zasilaniem bateryjnym, co również się nam przyda.
7. Raspberry Pi lub inny SBC.
8. Nie mam pojęcia, co jeszcze, czarować będziemy na bieżąco.

To było wstępne podsumowanie, czego możecie się spodziewać. Jeśli nie macie jakiejś płytki, nie powinno Was to martwić, ponieważ na każdej, którą znajdziecie na dnie szuflady, osiągniecie to samo, a może nawet więcej. Dla mnie jednak podczas tej wyprawy istotne jest zarówno wygodne operowanie przewodami, jak i kierowanie Waszymi umysłami.

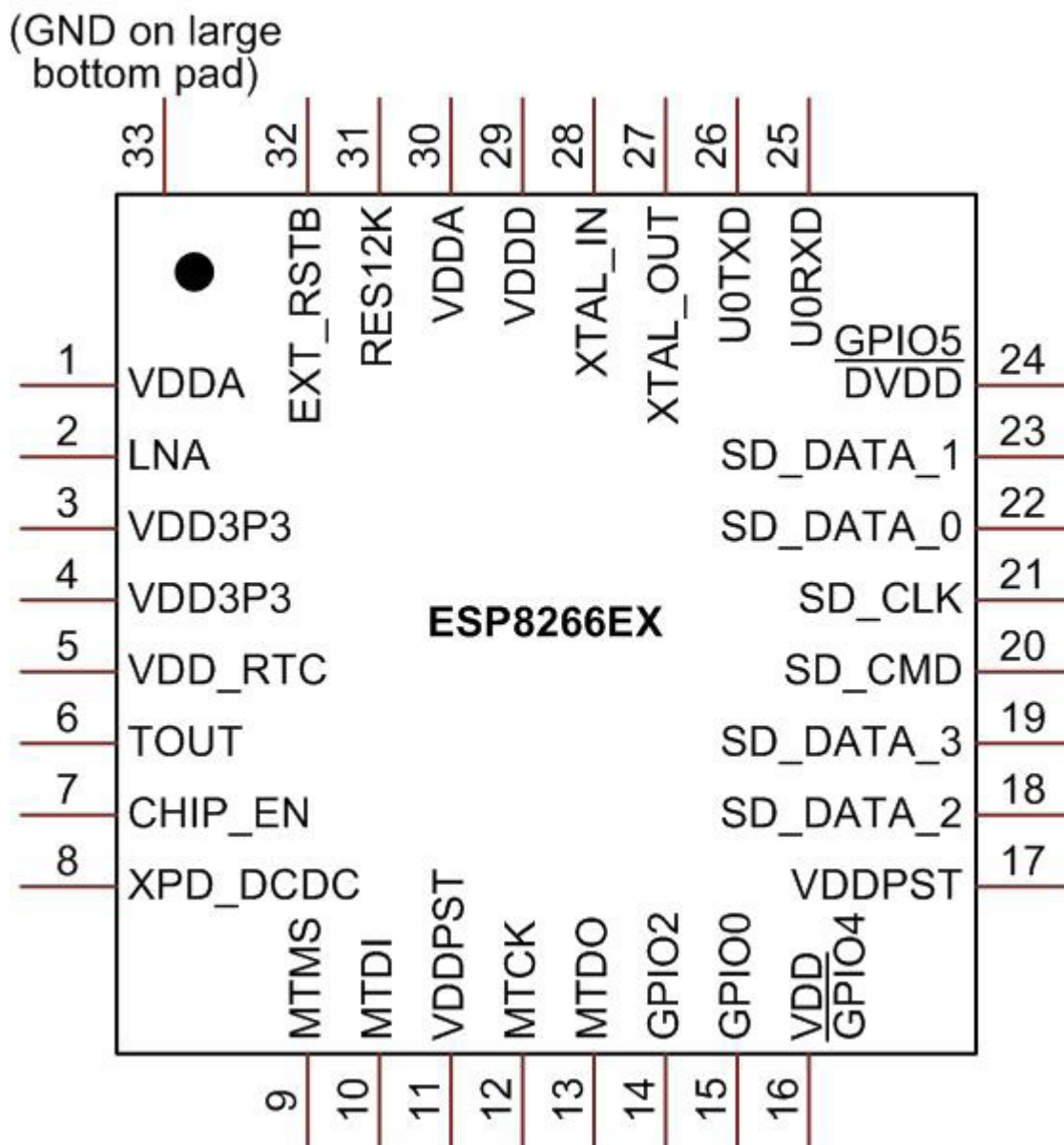
Jak zapewne zauważyliście, każda z płytek jest inna, inaczej wyposażona i oparta na innym układzie ESP. Ma to jednak sens, gdyż właśnie dzięki temu skakaniu po różnych układach nauczycie się je odróżniać i dobierać do własnych projektów. Wiedza ta pokaże Wam, że ESP-01 wprawdzie jest w porządku, ale nie nadaje się do niektórych rzeczy ze względu na ubogość dostępnych GPIO, i nikt Wam nie wmówi, że wersja ze specjalnym firmware jest tym, co Wam będzie potrzebne, i najlepszym rozwiązaniem. Tymczasem mając już rozeznanie w różnych modułach, chwycicie za inny, np. ESP-12, w którym jest więcej GPIO i który będzie pasować do Waszego pomysłu. Oczywiście tylko Wy wiecie, co będzie pasować do Waszego wynalazku.

Postaram się zanadto nie przynudzać, ale niestety bez garści technikaliów się nie obejdzie, podobnie jak bez doboru środowiska. Ale razem damy radę. Wywód postaram się poprowadzić z typowym dla mnie humorem, a niektóre

zagadnienia, wykraczające poza zarys, omówimy bardziej z grubsza, traktując je jako ciekawostki, warto mieć bowiem świadomość ich istnienia. Przyda się też praca z dokumentacją i schematami. Oczywiście nie znajdziecie tu tłumaczeń not producenta. Postaram się jednak pokazać programowanie i budowę urządzeń IoT jako dobrą zabawę i rozrywkę, przekazując przy okazji trochę wiedzy.

ESP8266 — charakterystyka układu

ESP8266 to był pierwszy dostępny w sprzedaży układ. Charakteryzuje się niewielkim rozmiarem, 5×5 mm, ale też skomplikowanym układem wewnętrznym, który ma dość dużą moc obliczeniową jak na takie maleństwo. Jego pinout jest pokazany na rysunku 2.1. To układ typu SoC (ang. *system-on-a-chip*). Zawiera w sobie cały zintegrowany system — radio WiFi, wzmacniacz, układy peryferyjne, pamięć RAM, 32-bitowy rdzeń Tensilic...



Rysunek 2.1. Pinout układu ESP8266EX²

Co prawda ESP8266EX jest bardzo prosty w zastosowaniu, nie ma dużych wymagań co do liczby elementów zewnętrznych i jest bardzo tani, względnie mało osób decyduje się użyć samego układu. Powodem jest dostępność ogromnej liczby gotowych płytek modułów, które wprowadziła na nasz rynek firma Ai-Thinker i które zawierają zarówno SoC, jak i konieczne elementy

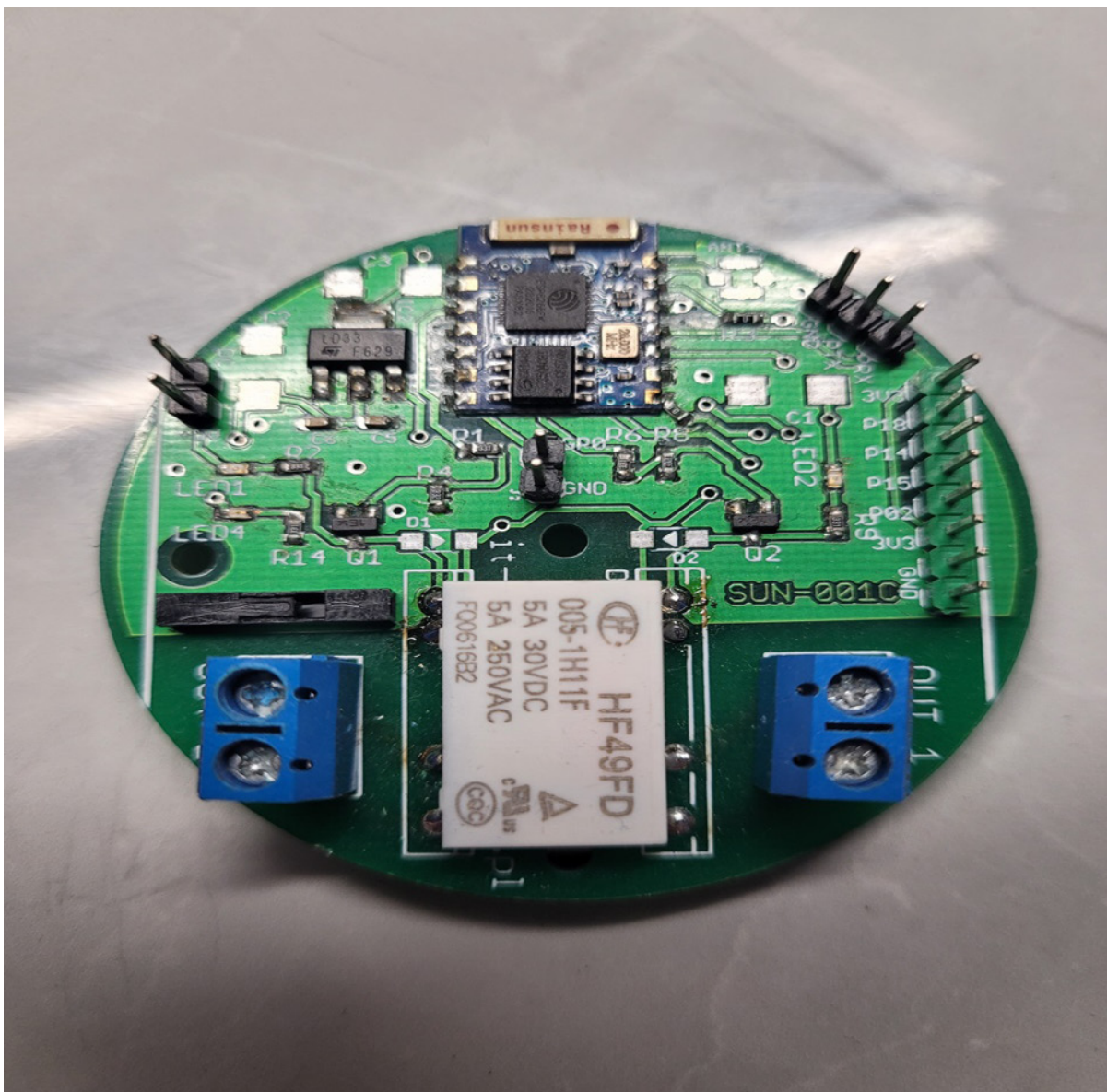
zewnętrzne, w porównaniu z innymi rozwiązaniami wygodne w użyciu i również tanie. Jest też dostępnych mnóstwo wariacji na płytkach uniwersalnych, dzięki którym wystarczy podłączyć zasilanie przez USB, i wszystko jest gotowe do pracy. Można nawet programować bez użycia zewnętrznych programatorów.

Jest tak dlatego, że domyślnie programowanie zewnętrznego flasha QSPI odbywa się za pomocą oddzielnych pinów UART poprzez bootloader. To proste rozwiązanie jest dość wygodne, szybkie i tanie, a dzięki temu łatwiej dostępne dla amatorów. Co właściwie dostajemy w opakowaniu?

Na przykład moduł ESP12F ma następujące parametry:

- napięcie zasilania: 3,3 V,
- pobór prądu w spoczynku: ok. 10 μ A,
- chip: ESP8266EX,
- pamięć flash: 4 MB,
- standard: WiFi 802.11 b/g/n,
- częstotliwość WiFi: 2,4 GHz,
- obsługa zabezpieczeń WPA/WPA2,
- możliwość działania w trybie AP (ang. *access point*), STA (ang. *standalone*) oraz AP+STA,
- obsługa poleceń AT,
- komunikacja: UART (Rx, Tx),
- domyślna prędkość UART: 115 200 b/s,
- montaż SMD lub THT,
- raster wyprowadzeń: 2 mm,
- średnica otworu: 0,9 mm,
- 16 wyprowadzeń w rastrze 2 mm,
- 11 $\times$ GPIO — wyjścia/wejścia cyfrowe,
- 1 $\times$ wejście ADC — przetwornik analogowo-cyfrowy,
- interfejsy komunikacyjne: SPI, UART, SDIO,
- obsługa oprogramowania Lua,
- wymiary płytki: 24 $\times$ 16 mm.

Całkiem sporo jak na takie maleństwo. Oczywiście między różnymi rodzajami modułów występują różnice w postaci rozmiaru modułu, rodzaju anteny (PCB lub ceramiczna), obecności złącza SMA, liczby wyprowadzonych pinów czy rozmiaru pamięci flash. Ten ostatni czynnik nie ma wielkiego znaczenia, bo nawet najmniejsze, najbardziej ubogie moduły zawierają co najmniej 512 KiB tej pamięci, co przy cenie paru złotych za całość jest ogromną ilością, a ponadto zawsze można ją wymienić, bo układ obsługuje do 16 MiB, czyli z dużym okładem. Ze względu na prostotę i łatwość użycia warto w tym miejscu wspomnieć o społeczności SUPLA, obsłudze LUA, Arduino i tak dalej. Przykładem układów opartych na ESP8266 jest moja mała płytka, nazwijmy ją umownie CircleBoard (rysunek 2.2).



Rysunek 2.2. CircleBoard

To jeden z pierwszych prototypów i ma już swoje lata. Pomyślano ją tak, by dała się zmieścić do puszek i była maksymalnie uniwersalna. Na pokładzie ma dwa przekaźniki i cztery GPIO, a ponadto złącze zasilania 5 V i złącze UART do programowania. I kilka drobiazgów niemających teraz znaczenia, ponieważ jeszcze do niej wrócimy. Teraz przede wszystkim chciałem pokazać, że na

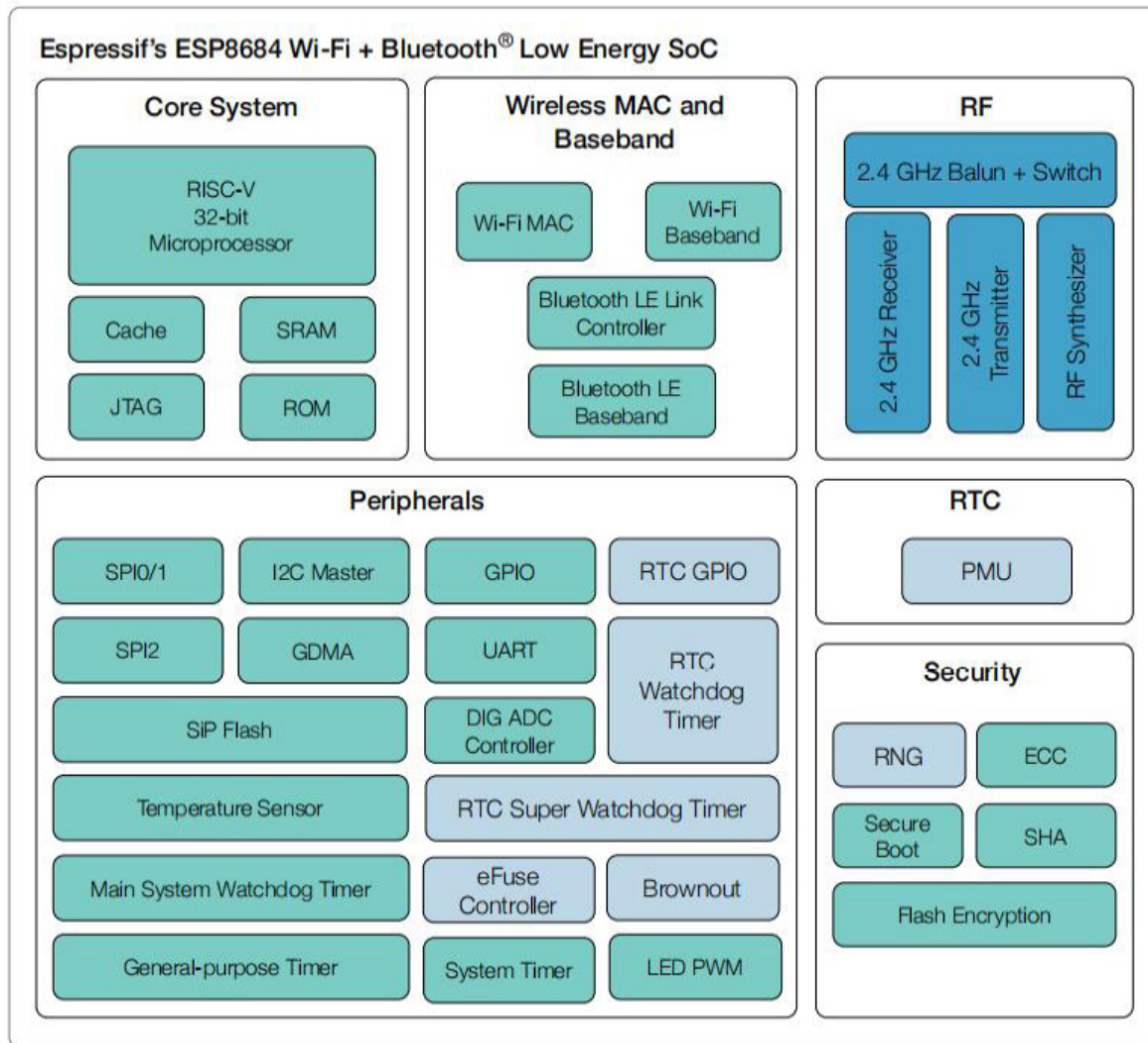
tak banalnym fundamencie możemy budować całą masę użytecznych urządzeń, takich jak:

- sterowniki rolet,
- sterowniki bram,
- sterowniki oświetlenia,
- sterowniki ogrzewania,
- sterowniki powyższych uwzględniające warunki, np. z czujników temperatury, oświetlenia, wilgotności.

Oczywiście to tylko wybrane, bo ogranicza nas tylko wyobraźnia, ale sami zobaczcie, jak łatwo, prosto i przyjemnie można zrealizować wiele projektów.

Na zakończenie tej krótkiej prezentacji warto wspomnieć, że 8266 doczekał się młodszego rodzeństwa w postaci następujących wariacji:

- **8285** — układ zawiera wbudowany 1 MiB pamięci flash i stanowi rozwiązanie w stylu *System-on-a-Chip*. Zapewnia też bogatszą listę dostępnych interfejsów.
- **8684** — główną zmianą jest zmiana rdzenia na opensource'owy 32-bitowy RISC-V (rysunek 2.3).



Rysunek 2.3. Schemat blokowy SoC ESP8264³

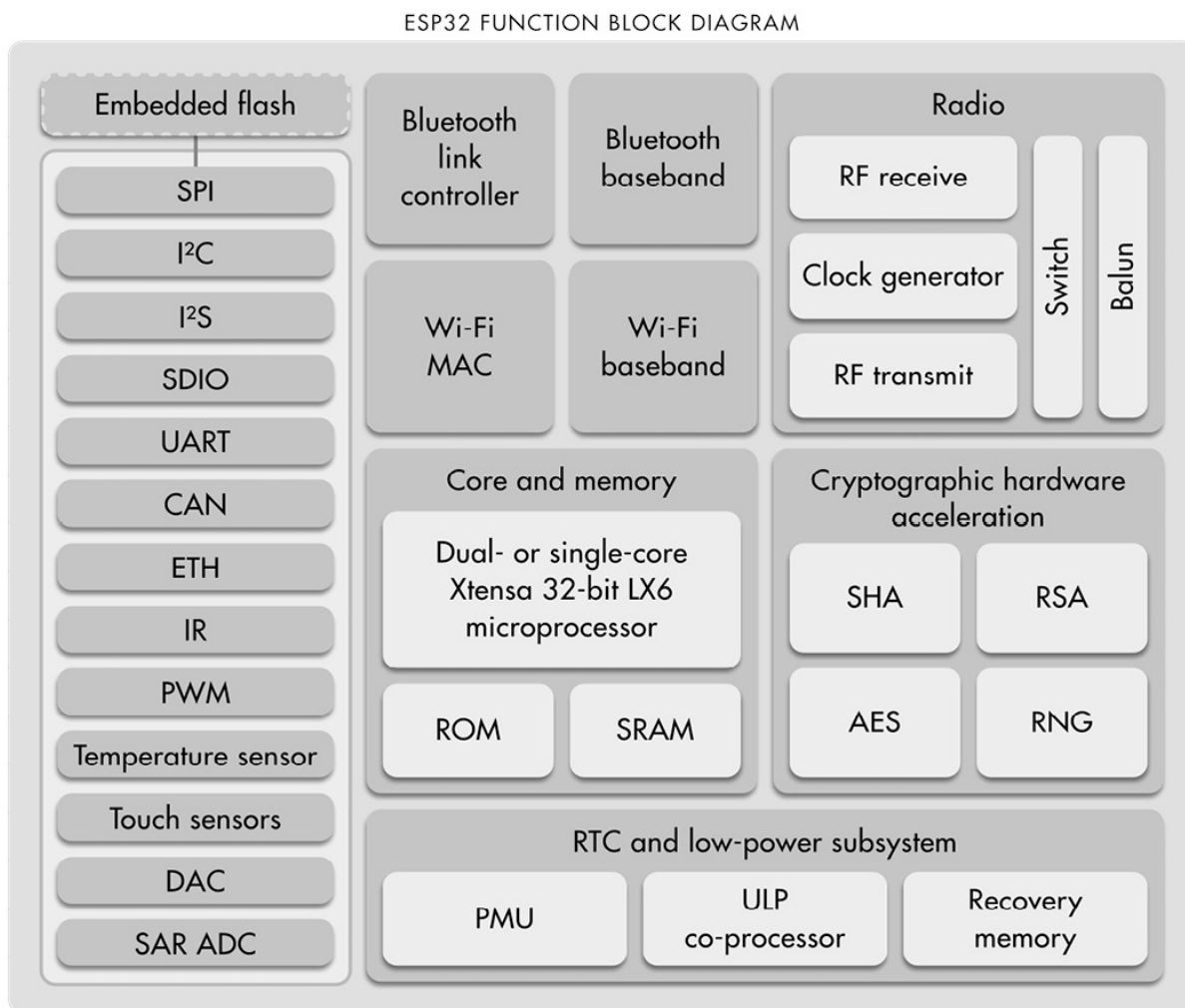
Jak widać, układ jest bogato wyposażony w peryferia. Spora zaletą jest też zewnętrzny układ pamięci flash, w niektórych przypadkach możliwy do samodzielnej wymiany na większy.

ESP32 — im dalej w las...

ESP32 to prawdziwa bestia. Mój pierwszy moduł z tym potworkiem to ESP-Wroom-32, dla którego

zaprojektowałem również kilka płytek, wciąż dostępnych w zaprzyjaźnionym sklepie elTy.pl. Niemniej zabawę z nim polubiłem szczególnie za sprawą zestawu ESP-Wrover-KIT firmy Espressif System. To potężna, choć już starawa, płytka, która jako pierwsza zawierała programator/debuger przeznaczony specjalnie do układów ESP32, oparty na układzie FT2232 i obecnie dostępny jako ESP-PROG. Główną różnicą między ESP8266 a ESP32 jest dostępność dwóch rdzeni, co miało na celu znaczne przyspieszenie operacji, gdy jeden rdzeń zajmował się programem, a drugi obsługiwał radio. Zapewne miało to w dużej mierze wyeliminować efekt zacinania się układu, spowolnienia działania i rozłączania z siecią. Na tym etapie ważne jest jednak to, że rdzenie były dwa, do tego 32-bitowe, i miały do dyspozycji sporo pamięci RAM i jeszcze więcej flash, do tego były taktowane z częstotliwością 240 MHz i miały do dyspozycji radio WiFi i na dokładkę Bluetooth 4.2, a wszystko to w cenie poniżej czołowych, popularnych na naszym rynku, głównie za sprawą tanich narzędzi, 8-bitowych układów śp. firmy ATMEL (obecnie Microchip).

Żeby nie ograniczać się do tak lakonicznego opisu, zerknijmy na schemat blokowy podstawowego ESP32, pokazany na rysunku 2.4.



Rysunek 2.4. Schemat blokowy SoC ESP32⁴

Imponujące, prawda? A spójrzcie tylko na spis najważniejszych cech:

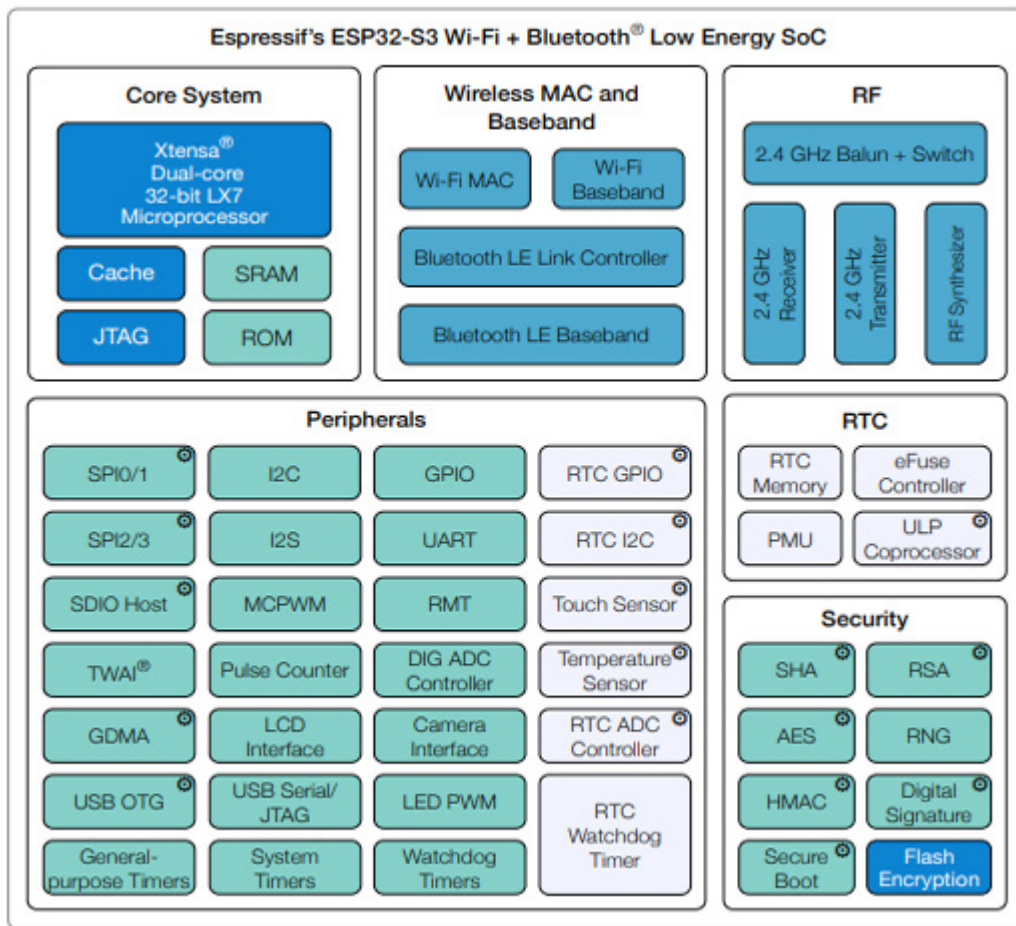
- 2 × Tensilic Xtensa 32Bit LX6,
- FCPU: maks. 240 MHz,
- wydajność: do 600 DMIPS,
- wbudowany koprocesor ULP pozwalający na konwersję ADC, obliczenia i wiele innych podczas głębokiego uśpienia,
- WiFi: 802.11 b/g/n 2,4 GHz, do 150 Mb/s,
- Bluetooth: v4.2 BR/EDR i BLE,

- pamięć ROM: 448 KiB,
- SRAM: 520 KiB:
 - Fast SRAM RTC — 8 KiB,
 - Slow SRAM RTC — 8 KiB,
 - eFuse — 1 KiB,
- wbudowana pamięć flash — 2 MiB w układach ESP32-D2WD, 4 MiB w układach ESP32-PICO-D4 SiP module, 0 MiB (ESP32-D0WDQ6, ESP32-D0WD i ESP32-S0WD — flash zewnętrzny),
- zewnętrzny flash z interfejsem QSPI, maks. 16 MiB,
- zewnętrzny SRAM z interfejsem QSPI, maks. 8 MiB,
- dostępne peryferia: ADC, I2C, DAC, UART, CAN2.0, SPI, I2S, RMII, PWM, z których większość współpracuje z DMA,
- zabezpieczenia:
 - IEEE 802.11 WPA, WPA/WPA2 i WAPI,
 - Secure Boot,
 - szyfrowanie pamięci flash (ang. *flash encryption*),
 - 1024-bitowy OTP, maks. 768 bitów dla użytkownika,
 - sprzętowy akcelerator kryptograficzny AES, SHA-2, RSA, krzywe eliptyczne (ECC), generator liczb losowych (RNG).

Jak widzicie, to całkiem sporo jak na mały i tani układ SoC. Oczywiście to nie wszystko. Z biegiem czasu wiele się zmieniło: pojawiło się wiele nowych układów, ulepszonych i wyposażonych w liczne nowości. Doszła np. łączność ZigBEE i wprowadzono lepsze wersje z jednym rdzeniem oraz wersje z rdzeniami RISC-V i ARM Cortex. Zwłaszcza RISC-V wnosi tu wiele nowego, a że jest open-source'owy, nie jest objęty skomplikowanymi zapisami licencyjnymi, jak Tensilic czy ARM.

Kolejną ciekawą pozycją jest nowy zestaw deweloperski SC01 PLUS, którego nazwa pewnie zabrzmiała dla Was tajemniczo (może poza bywalcami naszego forum). Jest

to zestaw firmy PanLee zawierający moduł ESP32-S3-WROOM-1-N16R2, którego schemat blokowy widzicie na rysunku 2.5. Czyli względem podstawowego ESP32-WROOM-32 wyposażono go w łączność Bluetooth 5.0, interfejs LCD, 2 MiB PSRAM i 16 MiB pamięci flash. Zestaw ten jest naprawdę potężny i w tej cenie nie kupicie nic innego z pojemnościowym dotykowym wyświetlaczem 3,5". Płytką tą jest idealna do zastosowań IoT. Pod względem wydajności i możliwości komunikacyjnych jest doskonałym wyborem do wykonania panelu operatorskiego, a dodatkowym ułatwieniem podczas projektowania GUI są wydajne biblioteki graficzne LV LVGL (ang. *Light and Versatile Embedded Graphics Library*), które pozwalają na tworzenie atrakcyjnych graficznie ekranów sterujących. Zestaw ten jest pokazany na rysunku 2.6.



Rysunek 2.5. Schemat blokowy ESP32-S3⁵

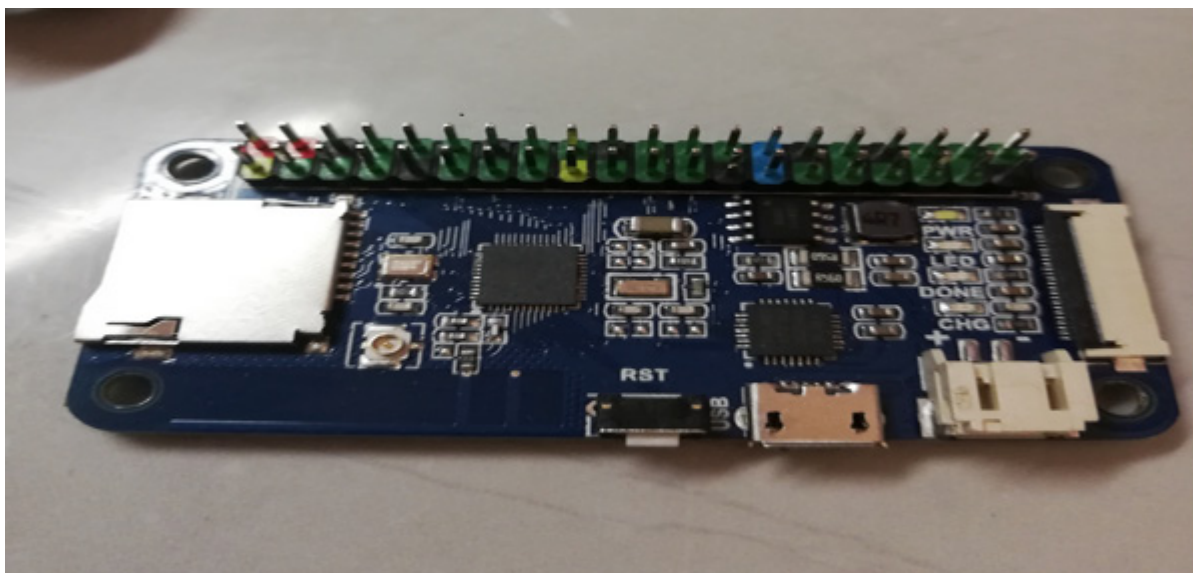


Rysunek 2.6. Zestaw PanLee SC01 Plus

Wspominam o innych układach i wersjach dlatego, że warto wiedzieć, jakie są możliwości, co pozwala lepiej dobrać rozwiązanie, w tym peryferia, do potrzeb. Wiedza ta może się okazać istotna podczas projektowania urządzenia albo budowania prototypu, czy wręcz kluczowa na etapie wdrażania nowego produktu.

Teraz nieco bliżej przyjrzymy się płytce ESP32 One (rysunek 2.7). To konstrukcja ciekawa o tyle, że rozkład elementów jest w formie popularnego Raspberry Pi Zero. SoC to dosłownie podstawowy ESP32. Ponadto mamy złącze kamery OV2640, 8 MiB pamięci PSRAM, gniazdo kart mikro-SD, układ CP2102 do programowania i komunikacji USB i tylko 4 MiB pamięci flash, jest też złącze

zasilania z pakietu Li-Ion i układ ładowania. Płytkę udostępnia ponadto mikrofon cyfrowy. Więcej informacji znajdziecie na stronie producenta pod adresem <https://www.waveshare.com/esp32-one.htm>.



Rysunek 2.7. Płytkę ESP32 One⁶

Jeśli chodzi o stronę sprzętową i terminologię, myślę, że na razie to wszystko, co powinniście wiedzieć. W dalszej części książki będę nawiązywał do tych informacji, ale też niektóre opisy skracał i upraszczał. Gdybyście mieli jakieś pytania, możecie — do czego zachęcam — zadać je choćby na wspomnianym forum.

ESP-PROG — gwarancja dobrej zabawy i bezpieczeństwa

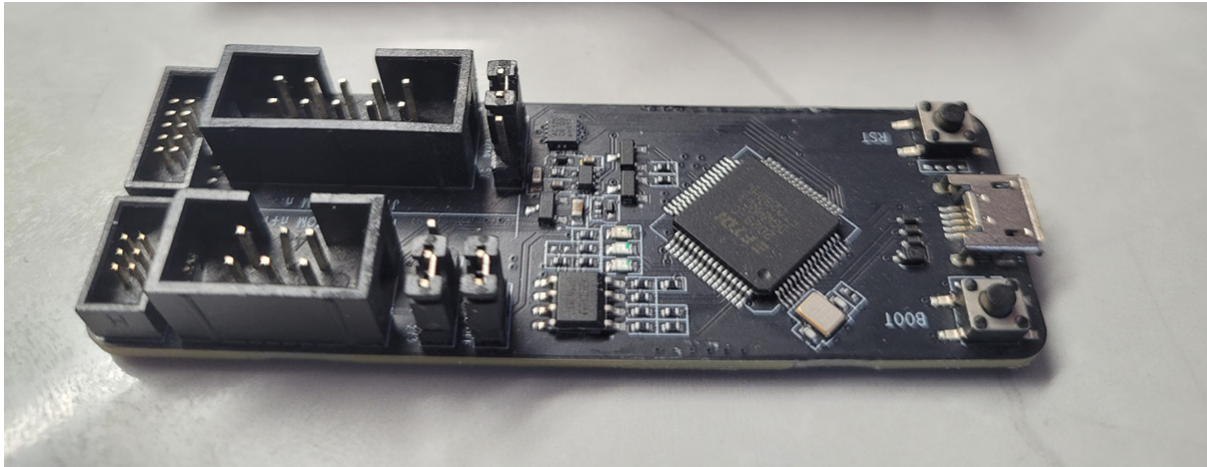
Jak wspominałem, programator/debuger nie będzie nam specjalnie potrzebny, ale jeśli go użyjemy, to w zasadzie tylko dla bezpieczeństwa, a właściwie jego poczucia. Jak już

pisalem, zarówno ESP8266, jak i ESP32 programujemy przez bootloader dowolnym konwerterem USB-UART. Najbardziej popularne konwertery tego typu to:

- FT232 — szybki i niezawodny układ firmy FTDI,
- CH340 — tani i prosty moduł chińskiej firmy, o dziwo bardzo sprawny,
- CP2102 — inny dobry układ USB UART.

Oczywiście nie musicie od razu biec do sklepu za rogiem czy nerwowo przeszukiwać internetowe fora, ponieważ znakomita większość dostępnych płytek z ESP8266 czy ESP32 ma na pokładzie taki konwerter. Najczęściej jest to CP2102 lub CH340. Rzadziej można natknąć się na układy FTDI, ale to dlatego, że ze względu na cenę zwykle trafiają do droższych produktów, w tym poważnych zestawów developerskich, takich jak wspomniany ESP-Wroover-KiT czy ESP32-S2-KALUGA-1. Jednak nawet spośród niedrogich płytek mało która nie jest wyposażona w jakikolwiek konwerter USB-UART. Owszem, zdarzają się takie jak choćby ESP-01, ale ten jest już bardzo stary i zaprojektowano go jako UART-WiFi sterowany poleceniami AT. Same moduły ESP, jak choćby ESP 12F czy też mój ulubiony ESP-03, który charakteryzuje się niewielkim rozmiarem i wbudowaną ceramiczną anteną, widoczny na rysunku 2.2, może kiedyś będziecie kupować w celu konstruowania własnych płytek, ale różnorodność dostępnych gotowych rozwiązań jest tak duża, że zdecydowana większość poprzestanie na nich, a wszelkie dostępne gotowe płytki z układami lub modułami ESP, takie jak NODE MCU czy WEMOS, po prostu podłącza się do gniazda USB, i już można się bawić w czary. Dlaczego więc wspominam o ESP-PROG-u? Uważam, że programatorów nigdy za wiele, a poza tym ESP-PROG to nie tylko programator, jak te znane Wam z mikrokontrolerów AVR; to przydatne narzędzie zawierające USB-UART, układ

sterowania wejściem w trybie bootloadera i pełnoprawny JTAG (rysunek 2.8).



Rysunek 2.8. ESP-PROG

Nie jest on drogi, a jest bardzo użyteczny i jak to się mówi, warto mieć go pod ręką. W przypadku własnych płytek, takich jak CircleBoard (rysunek 2.2), umożliwi nam programowanie układu. Czasem też w rozbudowanych programach bez JTAG-a trudno znaleźć błędy, zwłaszcza gdy rdzeń ciągle odmawia posłuszeństwa ☺.

Jeśli chodzi o stronę sprzętową, to chyba wszystko, co istotne. Całą resztę już na pewno macie — wyświetlacze, kabelki, czujniki... Pozostaje nam zatem przejść dalej.

Pamięci flash w układach ESP

Nasze układy, jak już wspominałem, używają zewnętrznych kości pamięci flash pracujących na magistrali SPI. W przypadku modułów, które nie mają blaszki, układ jest widoczny i w razie potrzeby łatwo go wymienić na większy. Z blaszką jest nieco trudniej. W każdym razie warto wiedzieć, że pojemność układów jest podawana w

megabitach (Mb), a nie megabajtach (MB). Stosuje się następujące układy:

- **2 Mb** (256 kB) — coraz rzadziej,
- **4 Mb** (512 kB) — najpopularniejsza, głównie ESP-01,
- **8 Mb** (1024 kB),
- **16 Mb** (2048 kB),
- **32 Mb** (4096 kB),
- **64 Mb** (8192 kB),
- **128 Mb** (16 384 kB),
- **256 Mb** (32 768 kB) — największa obsługiwana (jak dotąd spotkałem się z nią tylko na dwóch modułach).

Pamięci Data Flash są rozpoznawane poprzez *Chip ID*, który ma format HEX 0xAABBCCDD, gdzie:

AA — zwykle przyjmuje wartość 00, która jest pomijana, a podczas odczytu *Chip ID* używane są tylko 3 bajty.

BB — oznacza prawdopodobnie organizację pamięci 2^{xx} .

CC — szybkość w megahercach, zwykle 40 (40 MHz).

DD — ID producenta, najczęściej C8 (GigaDevice).

Fizycznie więc najczęściej będziemy mieć na pokładzie układy o oznaczeniach:

- 0x1740C8 — GD25Q64B, 8 MB,
- 0x1640C8 — GD25Q32B, 4MB,
- 0x1540C8 — GD25Q16B, 2 MB,
- 0x1440C8 — GD25Q80, 1 MB,
- 0x1340C8 — GD25Q40, 512 kB,
- 0x1240C8 — GD25Q20, 256 kB.

Czasem można znaleźć i mniejsze, ale trafiają się już naprawdę rzadko:

- 0x1140C8 — GD25Q10, 128 kB,
- 0x1040C8 — GD25Q12, 64 kB.

Często też będziemy natrafiać na układy innych firm, dlatego też zrobimy sobie w tym momencie listę tych układów wraz z ich kodami *Chip ID*. Dzięki temu będziemy mieć pełny obraz tego, co jest dostępne w danym module, oraz w miarę potrzeby tego, co można czym zastąpić.

Winbond

- 0x1640EF — W25Q32, 4 MB,
- 0x1540EF — W25Q16, 2 MB,
- 0x1440EF — W25Q80, 1 MB,
- 0x1340EF — W25Q40, 512 kB.

BergMicro

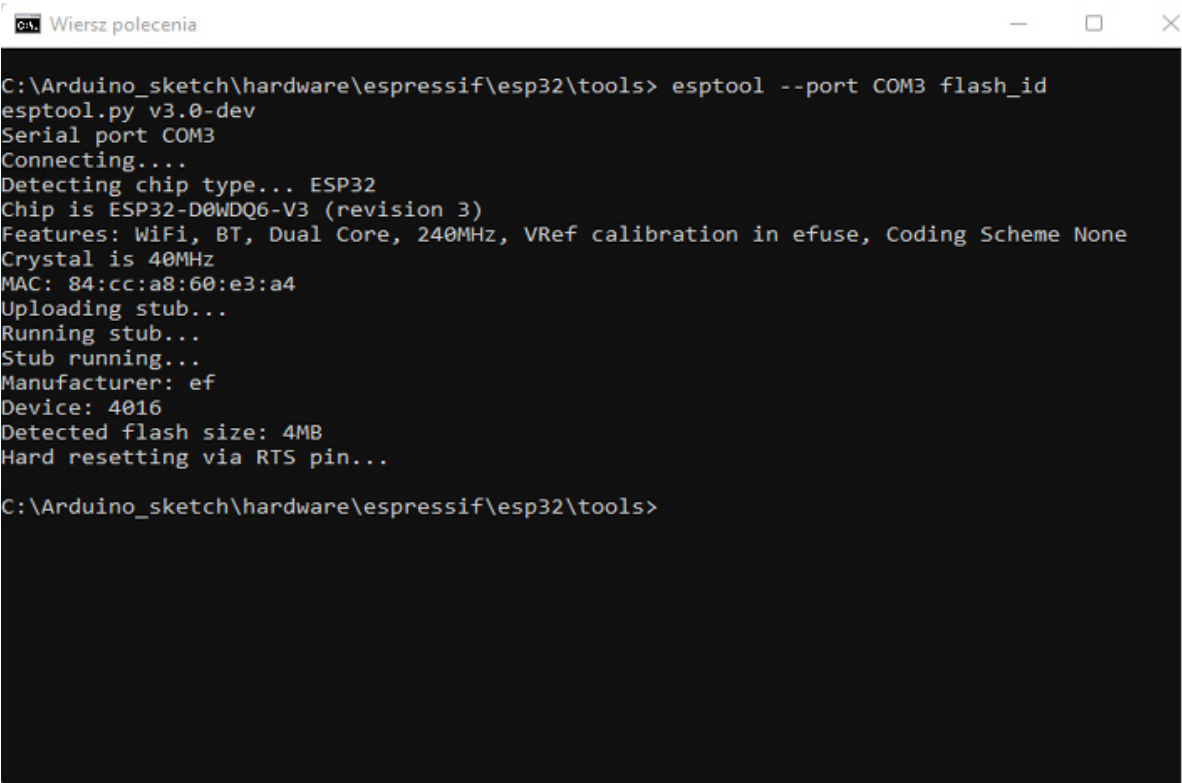
- 0x1640E0 — BG25Q32, 4 MB,
- 0x1540E0 — BG25Q16, 2 MB,
- 0x1440E0 — BG25Q80, 1 MB,
- 0x1340E0 — BG25Q40, 512 kB.

Takie oznaczenia widzimy, gdy moduł ESP8266 lub ESP32 nie jest zakryty blaszką, w przeciwnym razie — albo gdy, co gorsza, mamy już wzrok nieco popsuty od pracy z elektroniką — możemy łatwo, z użyciem narzędzia ESP Tool, wydobyć te informacje. Narzędzie to instaluje się wraz z pakietami dla płytek z ESP do Arduino i dla ESP-IDF. Są to dwa pliki w folderze *espXXX/tools*, czyli *esptool.py* i *esptool.exe*. Możemy je wykorzystać w narzędziu *Command Shell* (CMD).

Jak skorzystać z narzędzia? To proste i ogranicza się do kilku kroków:

1. Ustalamy port COM dla naszej płytki — np. *COM5*.

2. Uruchamiamy CMD.
3. Przechodzimy poleceniem **cd <ścieżka>** do folderu zawierającego ESP Tool. Dla przykładu, może to być: `cd C:\Arduino_sketch\hardware\espressif\esp32\tools`.
4. Będąc w folderze, wpisujemy polecenie do odczytu parametrów pamięci flash: `esptool --port COM3 flash_id` (rysunek 2.9).



```
C:\Arduino_sketch\hardware\espressif\esp32\tools> esptool --port COM3 flash_id
esptool.py v3.0-dev
Serial port COM3
Connecting....
Detecting chip type... ESP32
Chip is ESP32-D0WDQ6-V3 (revision 3)
Features: WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme None
Crystal is 40MHz
MAC: 84:cc:a8:60:e3:a4
Uploading stub...
Running stub...
Stub running...
Manufacturer: ef
Device: 4016
Detected flash size: 4MB
Hard resetting via RTS pin...

C:\Arduino_sketch\hardware\espressif\esp32\tools>
```

Rysunek 2.9. Działanie narzędzia ESP Tool

Tym sposobem dostajemy ogólne informacje na temat chipa (tu: ESP32) oraz interesujące nas dane dotyczące pamięci flash. W sumie najważniejsza dla nas informacja to jej wielkość, *flash size*, która w przypadku tego układu wynosi 4 MB, zastosowano więc układ o pojemności 32 Mb (4 MB, czyli 4096 kB).

ESP Tool to potężne narzędzie. To właśnie ono umożliwia programowanie i kasowanie pamięci flash naszych układów i nie tylko. Lista dostępnych opcji jest spora:

- `load_ram` — *Download an image to RAM and execute,*
- `dump_mem` — *Dump arbitrary memory to disk,*
- `read_mem` — *Read arbitrary memory location,*
- `write_mem` — *Read-modify-write to arbitrary memory location,*
- `write_flash` — *Write a binary blob to flash,*
- `run` — *Run application code in flash,*
- `image_info` — *Dump headers from an application image,*
- `make_image` — *Create an application image from binary files,*
- `elf2image` — *Create an application image from ELF file,*
- `read_mac` — *Read MAC address from OTP ROM,*
- `chip_id` — *Read Chip ID from OTP ROM,*
- `flash_id` — *Read SPI flash manufacturer and device ID,*
- `read_flash_status` — *Read SPI flash status register,*
- `write_flash_status` — *Write SPI flash status register,*
- `read_flash` — *Read SPI flash content,*
- `verify_flash` — *Verify a binary blob against flash,*
- `erase_flash` — *Perform Chip Erase on SPI flash,*
- `erase_region` — *Erase a region of the flash,*
- `version` — *Print esptool version.*

W tym miejscu przyznam się tak na marginesie, że celowo niektórych nazw czy opisów nie tłumaczę na polski, gdyż uważam, że techniczne sformułowania czy też nazwy i opisy funkcji są bardziej zrozumiałe w języku angielskim. Dodatkowo w ten sposób unikamy nieudomówień.

3 Źródło: https://www.mouser.pl/images/marketingid/2022/microsites/0/ESP8684_BlockDiagram.png; dostęp 20.06.2024.

4 Źródło: http://esp32.net/images/_resources/ESP32_Function_Block_Diagram.svg; dostęp 20.06.2024.

5 Źródło: <https://www.electronics-lab.com/wp-content/uploads/2023/01/esp32-s3-block-diagram-1.png>; dostęp 20.06.2024.

6 Źródło: <https://forum.lothar-team.pl/viewtopic.php?p=2778&hilit=ESP32+One#p2778>; dostęp 15.06.2024.

3. Interfejsy i peryferia

Podczas projektowania i tworzenia urządzeń nieuniknienie stykamy się z różnorodnymi interfejsami i peryferiami, które umożliwiają komunikację, sterowanie i zbieranie danych z różnych czujników. W tym kontekście używamy różnych metod komunikacji, protokołów i modułów, aby zrealizować konkretne funkcje swoich urządzeń.

W naszym labiryncie, gdzie każdy zakamarek może kryć nowe wyzwania, ważne jest zrozumienie różnych interfejsów, które będą używane. Jest to niezbędne do tego, by móc skutecznie poruszać się w tym gąszczu i nie błądzić, a naszym celem jest skonfigurowanie urządzeń zgodnie z wymaganiami. Wiedza ta pomaga nam uniknąć błędów, ułatwiając nawigowanie po rozmaitych aspektach projektu.

Część z kluczowych interfejsów, którymi się zajmiemy, obejmuje *General Purpose Input/Output* (GPIO), *Analog-to-Digital Converter* (ADC), *Universal Asynchronous Receiver-Transmitter* (UART), a także interfejsy komunikacyjne, takie jak I2C czy SPI. Istotne jest, by rozumieć ich działanie i zastosowanie, ponieważ są kluczowe dla naszej zdolności do skutecznego komunikowania się i współdziałania z otoczeniem.

W ten sposób, będąc świadomymi różnych narzędzi dostępnych w naszym "laboratorium" projektowym, możemy skonfigurować swoje urządzenia w sposób efektywny i zgodny z zamierzonymi celami. Wiedza ta jest niczym pochodnia w labiryncie: nie tylko daje nam ciepło, ale też pomaga rozwikłać zawiłości i skomplikowane zagadnienia, z którymi możemy się spotkać podczas wędrówki w ciemności. Dlatego też w tym miejscu

objaśnimy sobie to i owo, w taki sposób, żebyście mogli bez kłopotu zrozumieć to, o czym jest mowa dalej.

Digital GPIO

GPIO to skrót od *General Purpose Input/Output* (wejścia-wyjścia ogólnego przeznaczenia). Jest to standardowe oznaczenie pinów w mikrokontrolerach i innych układach, które mogą pełnić funkcję zarówno wejść, jak i wyjść, co oznacza, że mogą być używane do odczytywania sygnałów wejściowych (takich jak wciśnięcia przycisku) oraz sterowania sygnałami wyjściowymi (które mogą sterować np. diodami LED, silnikami, przekaźnikami). Czyli GPIO rozumiemy w kontekście różnych mikrokontrolerów, takich jak ESP32, jako piny, które mogą być konfigurowane jako wejścia lub wyjścia. Każdy pin *Digital GPIO* w przypadku wyjść może przyjąć stan wysoki (*HI*, 1) lub niski (*LOW*, 0), a w przypadku wejść odczytywać stan wysoki lub niski. To absolutna podstawa budowania interakcji z mikrokontrolerami.

Przykładowe zastosowania GPIO na mikrokontrolerze ESP32 obejmują:

- **wejścia cyfrowe** — do odczytywania stanu przycisków, przełączników czy sygnałów z czujników bazujących na sygnałach *HI/LOW*,
- **wyjścia cyfrowe** — do sterowania diodami LED, przekaźnikami czy innymi urządzeniami elektronicznymi.

Analog GPIO

Analog GPIO, zwykle nazywane pinami analogowymi, to specjalne piny wejścia-wyjścia w mikrokontrolerach, które

mają zdolność pomiaru sygnałów analogowych, czyli napięć zmieniających się płynnie w czasie. W przeciwieństwie do *Digital GPIO*, które działają w trybie cyfrowym (stan niski lub wysoki), piny analogowe mogą odczytywać zakres napięć w sposób ciągły. W mikrokontrolerach piny analogowe często oznacza się jako ADC (ang. *Analog-to-Digital Converter*) lub AIN (ang. *Analog Input*). Mogą być wykorzystywane do mierzenia napięć pochodzących z czujników analogowych, potencjometrów czy innych źródeł generujących sygnały o zmiennej wartości napięcia. Ich działanie charakteryzują następujące parametry:

- **Zakres pomiarowy.** *Analog GPIO* mają określony zakres pomiarowy, który zazwyczaj jest związany z napięciem od 0 V do maksymalnego napięcia zasilania mikrokontrolera (np. 3,3 V w przypadku ESP32). Działają one zazwyczaj na zasadzie podziału tego zakresu na pewną liczbę kroków, co determinuje rozdzielczość pomiaru.
- **Rozdzielczość.** Określa, na ile części zakresu pomiarowego jest podzielony sygnał analogowy. Na przykład mikrokontrolery często oferują ADC o 12-bitowej rozdzielczości, co daje nam 2^{12} (4096) możliwych wartości pomiaru. W przypadku ESP32 możemy ustawić rozdzielczości 9, 10, 11 lub 12 bitów. Im wyższa rozdzielczość, tym większa liczba możliwych wartości pomiaru, co przekłada się na dokładniejsze pomiary.
- **Precyzja.** Jest związana z dokładnością pomiarów i zależy od wielu czynników, takich jak jakość zasilania, stabilność napięcia referencyjnego ADC itp. W kontekście ESP32 oznacza dokładność pomiarów wykonanych przez jego wbudowany przetwornik analogowo-cyfrowy (ADC).

Napięcie referencyjne ADC

Precyzja pomiarów jest silnie związana z napięciem referencyjnym ADC, które jest stosowane do przeliczania napięć analogowych na wartości cyfrowe. Nasz ESP32 ma wbudowany regulator napięcia referencyjnego ADC, które wynosi 1100 mV (1,1 V). Oznacza to, że napięcia, które można bezpośrednio podawać na piny analogowe, mieszczą się w zakresie od 0 V do 1,1 V. Warto tu zaznaczyć, że nie należy podawać napięcia wyższego od 2,5 V, by nie doprowadzić do uszkodzenia ADC.

Konfigurowalność

ESP32 umożliwia skonfigurowanie wartości napięcia referencyjnego ADC w zależności od potrzeb projektu. Można to zrobić za pomocą funkcji `adc2_vref_to_gpio()`, która jako źródło napięcia referencyjnego pozwala ustawić dowolny pin GPIO. Stosowanie stabilnego i dokładnego napięcia referencyjnego wpływa na precyzję pomiarów.

Przykładowe zastosowania pinów analogowych obejmują pomiar napięcia z czujników temperatury, pomiar światła za pomocą fotorezystora czy odczytywanie pozycji potencjometrów. W kodzie programu odczyt z pinu analogowego można uzyskać za pomocą funkcji takich jak `adc1_get_raw`, co pozwala na konwersję analogowego sygnału na wartość cyfrową, którą można dalej przetwarzać w programie.

ADC

ADC (ang. *Analog-to-Digital Converter*) to elektroniczne urządzenie (lub komponent) w układzie, które przekształca sygnały analogowe na postać cyfrową. Sygnały analogowe są ciągłe w czasie i mogą przyjmować nieskończoną liczbę

wartości, natomiast sygnały cyfrowe są reprezentowane za pomocą skończonej liczby wartości cyfrowych. ADC jest często używany do pomiaru wartości sygnałów analogowych, takich jak napięcie elektryczne. Na przykład w przypadku pomiaru prądu przemiennego sygnał elektryczny, który jest reprezentowany jako ciągłe napięcie, może być podawany na wejście ADC. ADC przekształca ten sygnał na postać cyfrową, co umożliwia mikrokontrolerowi jego dalszą obróbkę, analizę i prezentację. W ESP32 dostępne są multipleksowane kanały ADC. Każdy kanał ADC może być używany do pomiaru napięcia na określonym pinie mikrokontrolera.

ADC jest niezmiernie istotnym narzędziem w elektronice, umożliwia bowiem mikrokontrolerom przetwarzanie informacji z otaczającego środowiska, takich jak wyniki pomiaru napięcia, temperatury czy prądu.

UART

UART (ang. *Universal Asynchronous Receiver/Transmitter*, asynchroniczny nadajnik-odbiornik) to popularny protokół komunikacyjny wykorzystywany do przesyłania danych między urządzeniami. Jest to protokół asynchroniczny, co oznacza, że dane są przesyłane między nadawcą a odbiorcą bez użycia wspólnego zegara. UART charakteryzuje się następującymi cechami:

- **Asynchroniczność.** UART działa w trybie asynchronicznym, co oznacza, że nie ma wspólnego zegara między urządzeniem nadającym a odbierającym. Zamiast tego każdy bajt danych jest poprzedzony i zakończony bitami startu i stopu.
- **Dwukierunkowość.** UART umożliwia komunikację dwukierunkową, co oznacza, że dane mogą być

przesyłane zarówno od nadawcy do odbiorcy, jak i od odbiorcy do nadawcy.

- **Prostota.** UART jest stosunkowo prosty do zaimplementowania i szeroko stosowany w różnych układach elektronicznych, np. mikrokontrolerach czy układach komputerowych.
- **Przesyłanie danych.** Szybkość transmisji danych (ang. *baud rate*) można dostosować do konkretnych wymagań komunikacyjnych. Najczęściej używane prędkości to 9 600, 19 200, 38 400, 57 600 i 115 200 bps (b/s).

UART jest szeroko stosowany w różnych dziedzinach, takich jak komunikacja między mikrokontrolerami, połączenia między komputerem a peryferiami (np. myszą, drukarką), komunikacja bezprzewodowa (Bluetooth, IrDA), a także w wielu innych systemach wbudowanych. W przypadku ESP8266, ESP32 czy Arduino UART używa się do programowania urządzenia oraz do komunikacji z innymi urządzeniami, takimi jak sensory, moduły GPS czy inne mikrokontrolery.

1-Wire

1-Wire to protokół komunikacyjny stworzony przez firmę Dallas Semiconductor (obecnie część Maxim Integrated), który umożliwia komunikację pomiędzy różnymi urządzeniami za pomocą jednego przewodu danych (ang. *one-wire*). Oczywiście wymagane są przewody zasilające VCC i GND, a jednoprzewodowość odnosi się do liczby przewodów danych. Ten prosty interfejs komunikacyjny jest często używany w zastosowaniach, w których trzeba efektywnie łączyć wiele urządzeń. Oto jego główne cechy:

- **Jednoprzewodowy interfejs.** Protokół 1-Wire korzysta z jednego przewodu danych (ang. *data line*), co

upraszcza układy połączeń i minimalizuje liczbę potrzebnych pinów na urządzeniach.

- **Pobieranie zasilania.** Wiele urządzeń obsługujących 1-Wire pozwala na zasilanie z linii danych, co oznacza, że nie wymaga osobnego przewodu zasilania. Urządzenia potrafią pobrać energię z impulsów na linii danych.
- **Unikalne adresy.** Każde urządzenie 1-Wire ma unikalny 64-bitowy lub 48-bitowy adres identyfikacyjny. Dzięki temu można łączyć ze sobą wiele urządzeń, a każde z nich będzie jednoznacznie identyfikowane.
- **Wiele urządzeń na jednej linii.** Protokół 1-Wire umożliwia łączenie wielu urządzeń na jednej linii danych, co pozwala na oszczędność pinów mikrokontrolera lub interfejsu komunikacyjnego.
- **Prędkość transmisji danych.** Standardowa prędkość transmisji danych w protokole 1-Wire to 16,3 kb/s, ale możliwe są również prędkości 142 kb/s i wyższe, w zależności od implementacji.
- **Różnorodność zastosowań.** Protokół 1-Wire jest szeroko wykorzystywany w różnych zastosowaniach, takich jak termometry cyfrowe (np. DS18B20), identyfikatory i karty pamięci (np. DS1990 — iButton) czy też kości pamięci EEPROM.
- **Komunikacja asynchroniczna.** Protokół ten jest asynchroniczny, co oznacza, że dane są przesyłane bez użycia stałego zegara i wymagają stosunkowo małej liczby linii komunikacyjnych.

Protokół 1-Wire jest ceniony za prostotę, niezawodność i niski koszt implementacji. Stosuje się go powszechnie w systemach kontroli i monitoringu, gdy trzeba efektywnie łączyć wiele urządzeń w jednej sieci.

I2C

I2C (ang. *Inter-Integrated Circuit*) to popularny dwukierunkowy, dwuprzewodowy protokół komunikacyjny stworzony do efektywnej komunikacji pomiędzy różnymi układami elektronicznymi. Opracowała go firma Philips Semiconductor (obecnie NXP Semiconductors) i dziś jest powszechnie używany w elektronice, zwłaszcza w środowiskach, gdzie potrzebna jest komunikacja między różnymi układami lub mikrokontrolerami. I2C oferuje m.in.:

- **Dwuprzewodowy interfejs.** I2C wykorzystuje tylko dwie linie komunikacyjne: linię danych (SDA — ang. *Serial Data Line*) oraz linię zegarową (SCL — ang. *Serial Clock Line*).
- **Dwukierunkowość.** Umożliwia przesyłanie danych w obie strony, co oznacza, że jeden układ może zarówno nadawać, jak i odbierać dane.
- **Szybkość transmisji.** Szybkość transmisji w I2C można dostosowywać w zależności od potrzeb danego układu, ale standardowe prędkości to 100 kb/s (*standard*), 400 kb/s (*Fast-mode*), 1 Mb/s (*Fast-mode plus*).
- **Urządzenia Master i Slave.** W komunikacji I2C jedno urządzenie pełni funkcję *master* (sterującego), podczas gdy pozostałe odgrywają rolę *slave* (podporządkowanych). Master inicjuje transakcje komunikacyjne.
- **Adresowanie.** Każde urządzenie I2C ma unikalny 7- lub 10-bitowy adres, co pozwala na skierowanie komunikatu do konkretnego urządzenia w magistrali I2C.
- **Bit startu i bit stopu.** Komunikacja w protokole I2C rozpoczyna się od bitu startu (ang. *start bit*) i kończy bitem stopu (ang. *stop bit*). Te bity sygnalizują rozpoczęcie i zakończenie transakcji.

I2C jest powszechnie stosowany w połączeniach między mikrokontrolerami, a także w komunikacji z różnymi peryferiami, takimi jak czujniki, ekrany LCD, pamięci

EEPROM czy różnego rodzaju układy wejścia-wyjścia. Dzięki swojej prostocie, elastyczności i wszechstronnemu zastosowaniu protokół I2C jest jednym z najpopularniejszych środków komunikacji między układami elektronicznymi w dzisiejszych systemach wbudowanych.

Podsumowanie

W labiryncie układów elektronicznych interfejsy są jak tajemnicze przejścia, którymi dane przemierzają ogrom korytarzy, tworząc puls codziennego życia cyfrowego. To nie tylko linie i piny, ale jakby dusze, które komunikują się między sobą niczym w motywach symfonii działających systemów. Możemy śmiało w naszym kontekście magicznego labiryntu abstrakcyjnie przyjąć, że:

GPIO to jak magiczne runy, które mikrokontrolery odczytują, interpretując je jako polecenia od użytkownika. To jakby fizyczne gesty układu, które czynią go interaktywnym i elastycznym.

UART to jak niewidzialne nici, które łączą różne elementy systemu w spójną całość. Jest to język komunikacji, w którym każdy impuls danych jest jak nuta w melodii cyfrowego koncertu.

I2C to jak sieć neuronów w układzie elektronicznego mózgu. Dane przenikają, układają się w struktury informacyjne, jakby budując wspólne myślenie systemu.

1-Wire to jak magiczne znaki, które czynią każde urządzenie niepowtarzalnym. Każde z nich jest jak postać w cyfrowej opowieści, a linia danych to jak most łączący różne światy w jedną, spójną historię.

Te interfejsy i peryferia możemy postrzegać jak klucze do wirtualnych bram, otwierające drogę do fantastycznych krain cyfrowych możliwości. W tym labiryncie interfejsy są jak

rozświetlone ścieżki, które prowadzą ku nowym pomysłom,
innowacjom i odkryciom. Dzięki nim dotrzemy bezpiecznie
do ukrytego wyjścia.

4. Bezpieczeństwo

Właściwie temat ten poruszam jako ciekawostkę. W przypadku sieci bezprzewodowych opartych na WiFi to na nas spoczywa obowiązek odpowiedniego ich zabezpieczenia przed nieupoważnionym dostępem. Układy ESP8266 i ESP32 dysponują oczywiście szyfrowaniem danych, takim jak:

- WEP,
- WPA/WPA2,
- PSK/Enterprise,
- AES/SHA2/Elliptical Curve Cryptography/ RSA-4096.

Ale bezpieczeństwo sieci WiFi to nie tylko kwestia zabezpieczeń. Są też bardziej złożone aspekty, które mogą być interesujące dla osób bardziej zaawansowanych technicznie. Na przykład moglibyśmy rozważyć scenariusze ataków takich jak *"man in the middle"* czy możliwość przechwycenia i analizy ruchu sieciowego. Oczywiście, dla przeciętnego użytkownika te bardziej złożone zagadnienia mogą nie mieć większego znaczenia. Niemniej warto mieć świadomość, że bezpieczeństwo sieci WiFi to coś więcej niż wpisanie hasła do sieci. To również zrozumienie, jakie zagrożenia istnieją i jak można się przed nimi zabezpieczyć. Mimo wszystko nie to jest sednem tego rozdziału. Celem naszych rozważań jest pewna sytuacja, która została wymyślona na potrzebę opisu metod zabezpieczeń układu ESP32.

Hipotetyczne rozważania

Założmy, że dany użytkownik — nazwijmy go Miętosław Koniuszko — zgłębił tematykę tej książki i wzbogacił wiedzę wskazanymi w tekście pozycjami, przeczytał też wiele innych książek i spędził godziny na próbowaniu swoich sił i eksperymentach, i wpadł na genialny pomysł. Poczuł bowiem, że właśnie odkrył świat nowych możliwości.

Postanowił stworzyć system sterowania typu smart dla domów i biur, ale że sam był rolnikiem, pomyślał, że rozszerzy system o moduły, które mogłyby monitorować i zabezpieczać te pierwsze, ale też pomóc mu w jego ciężkiej pracy na roli. Dniami i nocami kodował, testował, doskonalił swój system. W końcu, po wielu tygodniach ciężkiej pracy, udało mu się. Stworzył system, który był nie tylko skuteczny, ale także prosty w obsłudze i tani w produkcji. Całość potrafiła nie tylko włączać światła i monitorować temperaturę w domu, nie tylko kontrolować pomieszczenia biurowe i wejścia, ale też sterować pracą szklarni, zautomatyzować karmienie zwierząt i wiele innych... Gdy pochwalił się swoim systemem na spotkaniu koła gospodyń wiejskich, wielu sąsiadów patrzyło z zazdrością. Niektórzy chętnie by nabyli taki system i zainstalowali u siebie, a inni... no właśnie...

Sukces pana Koniuszki zrodził w nim pewne obawy. Obawiał się on, że jego kod, który stanowił serce systemu, mógłby zostać skradziony i powielony. Wiedział, że jego system był innowacyjny i że wiele firm ze Wschodu chciałoby mieć dostęp do jego kodu i powielać urządzenie. Oczywiście przez niską cenę wykonania i prostotę układu można by łatwo skopiować i powielić. Myślał więc, jak zabezpieczyć swoje dzieło, które działało już w sześciu wsiach. I nagle w siódmej wsi pojawił się odpowiednik, działający identycznie i do tego tańszy.

Historia Miętosława Koniuszki jest przypomnieniem i swoistą przestrogą, że innowacja i bezpieczeństwo idą ręka

w rękę zwłaszcza w czasach dzisiejszego postępu technicznego. Bez odpowiednich zabezpieczeń nawet najbardziej innowacyjny pomysł może stać się celem dla tych, którzy chcą go wykorzystać do własnych celów. Niestety układy ESP przez zewnętrzną pamięć flash są podatne na ryzyko utraty kodu. Oczywiście dla większości z nas będzie to bez znaczenia, niemniej warto wiedzieć, że można się obronić, choć wiąże się to z ryzykiem. Dlatego właśnie ten rozdział poświęcę opisowi zabezpieczenia układu, które udostępnia nam producent. Chcecie się dowiedzieć, czego nie wiedział pan Koniuszko?

FuseBity

Powinniście wiedzieć, że układy ESP udostępniają funkcje pozwalające na zabezpieczenie się przed dostępem do kodu osób niepowołanych. Jeśli mieliście już okazję używać innych mikrokontrolerów, np. układów z rodziny AVR, to w tej chwili na myśl przyjdą Wam FuseBity. Z myślą o mniej wtajemniczonych powiem w uproszczeniu, że są to swego rodzaju bezpieczniki. Można je porównać do wkładek topikowych czy, jeszcze prościej, do bezpieczników samochodowych. Ich ustawienie polega na przepaleniu bezpiecznika. Do ich ustawienia ("wypalenia") służy narzędzie `espfuse.py`. Jest przeznaczone do komunikacji z układami ESP, ale tylko w celu wypalenia **jednorazowo programowalnych** — powtarzam: **jednorazowo programowalnych!** — e-bezpieczników. Przez jednorazowość tej operacji rozumiemy, że wypalenie odbywa się wyłącznie w jednym kierunku, $0 \rightarrow 1$, nigdy odwrotnie. Czyli operacji nie da się cofnąć — jest nieodwracalna.



Ponieważ eFuse można zaprogramować tylko raz, za pomocą tego narzędzia można trwale uszkodzić lub "zablokować" układ ESP. Używaj go z dużą ostrożnością i na własne ryzyko!

W każdym razie `espefuse.py` od wersji 2.0 instaluje się razem z `esptool.py`. Większość bezpieczników w nowych układach jest "niespalona", czyli ma wartość "0". Pewnie zauważyliście, że napisałem "większość", więc się zastanawiacie, jakie mają ustawioną wartość "1". Otóż kilka jest już na etapie produkcji ustawianych właśnie na "1", czyli "spalonych". Są to:

- MAC address (ustawiany fabrycznie),
- kalibracja ADC,
- informacje o układzie i wersji,
- kilka innych, teraz nieistotnych.

Dlaczego więc takie coś mogłoby uszkodzić nam układ? Ano eFuse pozwala np. wyłączyć dostęp do UART, a tym samym uniemożliwić jakąkolwiek komunikację z układem, w tym możliwość odczytania/zapisania nowego kodu. Tym samym, jeśli był uruchomiony jakiś kod, to już tak zostanie na wieki. Można powiedzieć, że układ zabezpieczony nie pozwala na uzyskanie dostępu do pamięci flash zawierającej program, nie można w niej wykonać odczytu i zapisu. Wykonywany jest tylko zapisany w niej pierwotnie program.

Oczywiście jest to możliwość zabezpieczenia układu. Brak komunikacji uniemożliwia łatwe odczytanie *Chip ID*, *Flash ID* oraz zawartości pamięci flash. Ale też nie do końca, bo musimy pamiętać, że flash jest zewnętrznym układem pracującym na magistrali SPI, a to powoduje, że można go odczytać np. tanim programatorem CH341A czy innym układem. Rzecz jasna można się pokusić o zabezpieczenia fizyczne, np. zamiast używać modułu, możemy stworzyć własną płytkę, w której po lutowaniu będziemy mogli zeszlifować opisy układów i dodatkowo zalać je twardą

żywicą, tak że dostanie się do nich spowoduje trwałe uszkodzenie nie tylko płytki, ale i połączeń, a brak oznaczeń na układach dodatkowo utrudni wydostanie kodu i wydłuży czas tej operacji. Niemniej dalej można go wydostać i powielić, a nawet na jego podstawie napisać identyczny, więc to zabezpieczenie również nie jest doskonałe.

Wróćmy do `espefuse.py`. Obsługuje ono następujące polecenia:

- `dump`,
- `summary`,
- `burn_efuse`,
- `burn_block_data`,
- `burn_bit`,
- `read_protect_efuse` oraz `write_protect_efuse`,
- `burn_key`,
- `burn_key_digest`,
- `burn_custom_mac`,
- `get_custom_mac`,
- `adc_info`,
- `set_flash_voltage`,
- `execute_scripts`,
- `check_error`.

Szczegóły eFuse dla danego układu znajdziecie w dokumentacji producenta na stronie <https://www.espressif.com/en/support/documents/technical-documents>.

Czy jednak to oznacza, że w tej kwestii jesteśmy bezsilni? Nie do końca. Jest jeszcze coś, co nas może wspomóc w zabezpieczeniu się przed podobną sytuacją.

Szyfrowanie pamięci flash

Układy ESP, takie jak ESP32 i nowsze, oferują narzędzie do manipulowania danymi, które są związane z funkcjami bezpiecznego rozruchu (Secure Boot) i szyfrowania pamięci flash. Służy do tego celu polecenie narzędzia `espsecure.py`. Narzędzie to pozwala na zdalne podpisywanie kodu kluczem za pomocą zewnętrznego modułu HSM. Jednak żeby móc skorzystać z tej funkcjonalności, trzeba zainstalować ESP Tool z pakietem HMS, co zapewni dostęp do interfejsu PKCS, pozwalającego na komunikację z zewnętrznym modułem. W tym celu należy wydać polecenie Pythona:

```
pip install 'esptool[hsm]' esp_hsm_sign espsecure.py
```

Teraz wystarczy użyć polecenia:

```
python espsecure.py sign_data -version 2 -hsm -hsm-  
config <hsm_config_file> -output <signed_image>  
<datafile>
```

(Oczywiście zawartość nawiasów zmieniamy na nazwy plików właściwe dla naszego projektu).

Działa to tak:

1. Z modułu HSM zostaje wyodrębniony klucz publiczny.
2. Zostaje wygenerowany podpis dla obrazu z użyciem modułu HSM.
3. Zostaje utworzony blok podpisu, a następnie dołączony do obrazu w celu wygenerowania podpisanego pliku obrazu.

Oczywiście klucz nie musi być przechowywany w zewnętrznym HSM. Możemy samodzielnie określić argument `-pub-key`, dostarczając własny klucz:

```
python espsecure.py sign_data -version 2 -hsm -hsm-  
config <hsm_config_file> -pub-key <public_key> -output  
<signed_image> <datafile>
```

gdzie `<public_key>` oznacza nasz prywatny klucz.

W ESP-IDF możemy w konfiguracji włączyć w ustawieniach projektu funkcję `CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES`, co uniemożliwi nam wygenerowanie niepodpisanego pliku obrazu. Gdy już mamy wygenerowany podpisany obraz binarny, wykonujemy jego weryfikację:

```
python espsecure.py verify_signature -version 2 -hsm -  
hsm-config <hsm_config_file> <signed_image>
```

W przypadku zewnętrznego klucza spoza HSM musimy określić argument `-keyfile`, czyli dostarczyć swój klucz zewnętrzny:

```
python espsecure.py verify_signature -version 2 -  
keyfile <public_key> <signed_image>
```

Jak na pewno zauważyliście, będzie nam też potrzebny plik konfiguracyjny modułu HSM. W pliku tym muszą być wypełnione pola: (`pkcs11_lib`, `credentials`, `slot`, `label`) `label_pubkey`, które odpowiadają konkretnemu używanemu modułowi HSM. Uzyskanie dostępu do tokena HSM dla wybranego slotu wymaga podania PIN-u użytkownika tokena i spowoduje wyświetlenie monitu o konieczności jego wprowadzenia. Oczywiście można alternatywnie użyć pola `credentials` w pliku konfiguracyjnym do przechowywania PIN-u w postaci zwykłego tekstu, co zautomatyzuje proces podpisywania. Przykładowa zawartość pliku `hsm_config.ini`, która używa SoftHMSv2 (czyli wersji programowej) jako zewnętrznego modułu HSM, przedstawia się następująco:

```
# hsm_config.ini  
# Config file for the external Hardware Security Module  
# to be used to generate signature.  
[hsm_config]  
# PKCS11 shared object/library module of the HSM vendor  
pkcs11_lib = /usr/local/lib/softhsm/libsofthsm2.so
```

```
# HSM login credentials
credentials = 1234
# Slot number to be used
slot = 1108821954
# Label of the object used to store the private key
label = Private Key for Digital Signature
# Label of the object used to store corresponding
public key
# label_pubkey is the same as label if the label for
public key
# is not explicitly specified while key-pair
generation.
label_pubkey = Public Key for Digital Signature
```

(W przypadku Windows należy zmienić ścieżki).

Rzecz jasna szczegóły znajdziecie w dostępnej na stronach producenta dokumentacji, do której odnośnik zamieściłem wcześniej. Oczywiście poza szyfrowaniem flasha i eFuse jest jeszcze...

Secure Boot

To nic innego jak bezpieczny rozruch, który jest obsługiwany przez układy ESP32 i nowsze, takie jak C2/C3 itd. Obecnie zalecane jest używanie Secure Boot v2, który jest bezpieczniejszy i bardziej elastyczny niż v1, jednak starsze układy mogą go nie obsługiwać.

Secure Boot v2 wykorzystuje weryfikację aplikacji i programu ładującego opartego na RSA-PSS (program ładujący drugiego stopnia). Może być również używany jako odnośnik do klucza służący do podpisywania aplikacji z użyciem schematu RSA-PSS bez podpisywania programu

ładującego. Jeśli chcemy go użyć w konfiguracji ESP-IDF, w menuconfig musimy ustawić CONFIG_ESP32_REV_MIN na wartość równą Rev 3 lub większą. Secure Boot chroni urządzenie przed uruchomieniem nieautoryzowanego (niepodpisanego) kodu przez sprawdzenie, czy każdy uruchamiany fragment oprogramowania jest podpisany. W ESP32 te elementy oprogramowania obejmują program ładujący drugiego stopnia i plik binarny każdej aplikacji. Jednakże trzeba mieć na uwadze, że program ładujący pierwszego stopnia nie wymaga podpisu, ponieważ jest to kod ROM, dlatego nie można go zmienić.

Schemat weryfikacji Secure Boot oparty na RSA (Secure Boot V2) jest zaimplementowany w ESP32 (ECO 3 i nowszych). Cały zaś proces bezpiecznego rozruchu w ESP32 wykonuje następujące kroki:

1. Gdy program ładujący pierwszego stopnia ładuje program ładujący drugiego stopnia, weryfikowana jest sygnatura RSA-PSS programu ładującego drugiego stopnia. Jeżeli weryfikacja przebiegła pomyślnie, wykonywany jest program ładujący drugiego stopnia.
2. Gdy program ładujący drugiego stopnia ładuje określony obraz aplikacji, weryfikowany jest podpis RSA-PSS aplikacji. Jeżeli weryfikacja przebiegła pomyślnie, wykonywany jest obraz aplikacji.

Rozwiązanie to ma sporo zalet, które mogą być niekiedy zbawienne dla nas jako developerów oprogramowania, a mianowicie:

- Klucz publiczny RSA-PSS jest przechowywany na urządzeniu. Odpowiadający mu klucz prywatny RSA-PSS jest zaś przechowywany w tajnym miejscu (np. sejfie) i urządzenie nigdy nie uzyskuje do niego dostępu.
- Na etapie produkcji chipa można wygenerować i zapisać w nim tylko jeden klucz publiczny.

- Ten sam format obrazu i ta sama metoda weryfikacji podpisu są stosowane w przypadku aplikacji i programu ładującego oprogramowanie.
- Na urządzeniu nie są przechowywane żadne sekrety. Dlatego jest odporny na tzw. pasywne ataki *side-channel* (analiza czasu lub mocy itp.).

Istotną jednakże zaletą Secure Boot jest to, że można go używać niezależnie od funkcji szyfrowania pamięci flash, niemniej zaleca się, by w celu zmaksymalizowania bezpieczeństwa środowiska używać jednocześnie obu funkcjonalności. Przy czym należy pamiętać, że Secure Boot ogranicza możliwość aktualizacji oprogramowania układowego. Dlatego warto zajrzeć po szczegóły do dokumentacji.

Pewnie się zastanawiacie, co dalej i że trochę to wszystko zawile, ale muszę w tym miejscu wspomnieć, że można przygotować program ładujący umożliwiający ponowne wgranie oprogramowania układowego lub jego aktualizację. Konfiguracja ta jest zalecana dla urządzeń produkcyjnych. W tym trybie każde urządzenie otrzymuje unikalny klucz, który nigdy nie jest przechowywany poza fizycznym urządzeniem. Ustawia się to przez wybranie w konfiguracji opcji *SecureBoot: One-Time Flash*.

Oczywiście istnieje też tryb alternatywny, *CONFIG_SECURE_BOOTLOADER_MODE*, który umożliwia dostarczenie pliku z kluczem binarnym używanym jako klucz Secure Boot. Mając ten klucz, możemy wygenerować nowe obrazy bootloadera. Działa to tak, że podczas kompilacji ESP-IDF generuje 256-bitowy klucz na podstawie klucza podpisu aplikacji ECDSA, który wygenerowaliśmy. Skrót SHA-256 tego klucza prywatnego zostanie użyty jako bezpieczny klucz bootloadera i umieszczony w eFuse. Teraz, zależnie od wybranego schematu kodowania, będzie on występował w postaci

niezmienionej — opcja *NONE*, lub przycięty do 192 bajtów — opcja *3/4 encoding*. O dziwo, jest to wygodne rozwiązanie, pozwala bowiem wygenerować i chronić jeden klucz prywatny.

Włączenie bootloadera umożliwiającego ponowne wgranie obrazu binarnego lub aktualizację odbywa się w menu konfiguracji projektu, gdzie wybieramy opcję *CONFIG_SECURE_BOOT/CONFIG_SECURE_BOOTLOADER_MODE/Bootloader*

Config/Config_Secure_BOOT_V1_Enabled i ustawiamy na *Reflashable*. Następnie jest już z górki. Oczywiście, jeśli trzeba, ustawiamy też po drodze *CONFIG_SECURE_BOOTLOADER_KEY_ENCODING*, zależnie od schematu kodowania, który znajdziemy w wierszu *Features* zaraz po podłączeniu do układu przez *esptool.py* lub na wyjściu wydrukowanym przez polecenie *espefuse.py summary*. Teraz w kilku krokach raz-dwa załatwiamy sprawę:

- Generujemy klucz podpisu i w menu podajemy ścieżkę do pliku klucza (*Secure Boot Configuration*).

Przykładowo w OpenSSL możemy wygenerować klucz poleceniem konsoli:

```
openssl ecparam -name prime256v1 -genkey -noout -  
out my_secure_boot_signing_key.pem
```

- Uruchamiamy polecenie *idf.py bootloader*. Zostanie utworzony plik binarny klucza na podstawie podpisu z klucza prywatnego. Pojawią się dwa migające zestawy poleceń. Pierwszy, *espefuse.py burn_key*, zawiera polecenie do zapisania klucza bootloadera w eFuse — **działa ono tylko raz i nie można tego zmienić**. Drugi zaś, *secure_boot_v1 path_to/secure-bootloader-key-xxx.bin*, można wykorzystać do ponownego wgrywania obrazu programu bootloadera z pomocą wstępnie obliczonego podsumowania wygenerowanego podczas kompilacji.

W zasadzie na tym zakończymy ten opis stosowanych zabezpieczeń. Nigdy go nie stosowałem osobiście, niemniej warto wiedzieć, że są sposoby na zabezpieczenie się przed nieautoryzowanym dostępem do oprogramowania układowego. A w razie potrzeby łatwo znaleźć szczegóły w dokumentacji. Dlatego też zaznaczam, żeby nie traktować tego materiału jako wytycznych, lecz jedynie jako ciekawostkę, gdyż w swoich "nieprodukcyjnych" zmaganiach na pewno nie będziecie z nich korzystać.

5. Środowisko

Oczywiście dobry sprzęt nie wystarczy, każdy bowiem jest tylko tak — albo aż tak — dobry jak programista, który wykrzesze z niego życie. Tu należy się więc zastanowić, w jaki sposób spróbujemy okiełznać mały układzik i coś za ducha chcemy począć. Doszliśmy więc do etapu środowiska, w którym odbędą się pewne czary, kiedy to przelejemy swój pomysł w realne, żyjące urządzenie.

Nie, w tym miejscu nie będę Was zachęcał do używania jedynie słusznego środowiska, ponieważ takiego nie ma, albo raczej jest, ale dla każdego inne. Jeśli chodzi o ESP, pisać programy można w wielu językach oraz środowiskach i frameworkach:

- ESP-IDF — oficjalne SDK firmy Espressif dla serii SoC ESP32, ESP32-S, ESP32-C i ESP32-H,
- Arduino ESP8266/ESP32 — port Arduino dla ESP8266, ESP32, ESP32-S2, ESP32-S3 i ESP32-C3,
- Espruino — JavaScript SDK i oprogramowanie sprzętowe ściśle emulujące Node.js,
- MicroPython (i CircuitPython) — odchudzona implementacja Pythona 3 dla mikrokontrolerów,
- Moddable SDK — nowoczesny JavaScript i TypeScript z obsługą sieci, grafiką i interfejsami API ECMA-419,
- Mongoose OS — obsługujący podłączone do chmury mikrokontrolery małej mocy, takie jak ESP32 i ESP8266, programowalny w JavaScript lub C (platforma polecana przez Espressif Systems, AWS IoT i Google Cloud IoT),
- mruby dla ESP32,
- NodeMcu — oprogramowanie oparte na Lua,
- PlatformIO,

- Rust,
- Visual Studio Code z oficjalnie obsługiwanym rozszerzeniem Espressif Integrated Development Framework (ESP-IDF),
- Zerynth — Python dla IoT i mikrokontrolerów, w tym ESP32.

Jak widzicie, każdy znajdzie dla siebie coś wygodnego i fajnego. Ja wolę używać oficjalnego SDK, czyli ESP-IDF, i pisać w czystym C, lecz jak wspominałem, nikogo do niczego nie będę przekonywał. Dlatego też nie będziemy się sztywno trzymać IDF, ale będę używał popularnych środowisk i frameworków, tak by pokazać Wam, że złoty środek nie istnieje i że sposób, w jaki dążycie do celu, nie ma tak naprawdę znaczenia.

Tu może trochę cofnę się do wstępu, gdzie jak pamiętam, pisałem coś o kluczu do labiryntu. Tak... zdecydowanie tak. A więc, moi drodzy, zapamiętajcie to, co teraz przeczytacie, bo jest to bardzo ważne: kluczem jesteście WY! — tak, bo nikt inny, tylko WY będziecie wymyślać nowe urządzenia i nowe rozwiązania.

No okej, zatem zajmijmy się naszym problemem, czyli środowiskiem. Jak pisałem, będę używał, a może nadużywał, popularnych rozwiązań, a więc na pewno przewinie się kilka środowisk:

- Arduino IDE,
- Visual Studio Code,
- Visual Studio Community,

w których będę używał wszelkich dostępnych dla naszych układów metod programowania. Jak widzicie, wymieniłem trzy środowiska, z których każde ma swoje zalety i wady.

Pierwsze z wymienionych, Arduino IDE, jest dość proste, można by rzec siermiężne, jak to twory napisane w Javie,

ale zyskało wielkie uznanie amatorów na całym świecie. Dodatkowym atutem jest olbrzymia społeczność i wiele gotowych rozwiązań. Sam za nim nie przepadam, ale na potrzeby tej książki będę korzystał z frameworka Arduino w prostszych przykładach.

Drugie, Visual Studio Code, to lekka międzyplatformowa wariacja Microsoftu na temat Visual Studio. To rozwiązanie również zyskało licznych zwolenników i w zasadzie wydaje się dla nas optymalne, głównie za sprawą PlatformIO, które umożliwia wygodne tworzenie programów dla wielu mikrokontrolerów, w tym ESP8266 i ESP32, nie tylko pod kontrolą frameworka Arduino, ale również oficjalnego SDK i wielu innych. A więc to takie wszystko w jednym.

Ostatni punkt, Visual Studio Community, to prawdziwy "kombajn", który kojarzy się przede wszystkim z tworzeniem aplikacji desktopowych dla systemu Windows. Obecnie można tworzyć w nim także aplikacje dla Androida, Linuxa i macOS. Za sprawą zaś pluginów — również dla systemów *embedded*, a co za tym idzie — ESP8266 i ESP32.

Jak widzicie, wszystkie się łączą w jednym punkcie, choć nie do końca. Dwa pierwsze są bezpłatne, zarówno dla amatorów, jak i profesjonalistów. Oczywiście Visual Studio Community również jest bezpłatne, ale już pluginy pozwalające na współpracę z naszymi układami są płatne, co dla wielu może być nie do przyjęcia. Są dostępne dwa godne polecenia:

- VisualMicro — to plugin pozwalający na używanie frameworka Arduino w porządnym IDE wraz z wieloma udogodnieniami, których nie znajdziecie w oryginalnym środowisku. Bezpłatnie możecie testować do woli to narzędzie przez 45 dni; po tym czasie należy wykupić licencję. Wszystkie informacje można znaleźć na stronie projektu, pod adresem <https://www.visualmicro.com/>.

- VisualGDB — to już bardzo potężne narzędzie. Też pozwala na używanie frameworka Arduino, ale dodatkowo również, w bardzo wygodny sposób, oficjalnego SDK. Udostępnia wiele przydatnych funkcji i bardzo dobry debugger oraz funkcję IntelliSense. Plugin jest oczywiście płatny; informacje o kosztach są dostępne na stronie projektu pod adresem <https://visualgdb.com/>.

Dostępność wielu środowisk IDE oraz różnych pluginów powoduje, że na pewno każdy znajdzie coś dla siebie, by móc wygodnie pisać i tworzyć swoje cuda, dlatego też nie zamierzam Wam mówić, co jest najlepsze i jedynie słuszne. Tymczasem dotarliśmy do punktu, w którym możemy śmiało powiedzieć, że już mniej więcej wiemy, co jest czym i jak się do tego zabrać, więc możemy, nie przedłużając, przystąpić do przygotowania naszego komputera na wzór kociołka Panoramixa, w którym już niedługo będziemy mogli warzyć czarodziejskie mikstury swoich kodów i ożywiać środowisko wokół nas. I choć w istocie nie od razu Rzym zbudowano, tak też i my powoli wspólnie odkryjemy tajniki łączenia się z WiFi i proste sterowanie urządzeniami, aż po bardziej wyrafinowane metody, w których użyjemy brokera MQTT, czy to w chmurze, czy własnego. Do tej pory opisy były dość ogólnikowe i wielu z Was zapewne mocno się wynudziło, niemniej miało to więcej sensu, niż moglibyście pomyśleć. Tym sposobem w lekki sposób przybliżyłem Wam, z czym się będziemy borykać. Przed nami jeszcze trochę nudy, ale już powoli będziemy konkretyzować wszystko, o czym dotąd pisałem. Zanim bowiem zajmiemy się właściwymi czarami, musimy wszystko uporządkować i przygotować, musicie też dowiedzieć się jeszcze kilku rzeczy. A potem pozostanie już czysta magia. Jeśli dotrwaliście aż do tego miejsca, to śmiało mogę powiedzieć, że pierwszy krok już zrobiliście. Teraz czas na drugi...

CZĘŚĆ II. Przygotowania do programowania

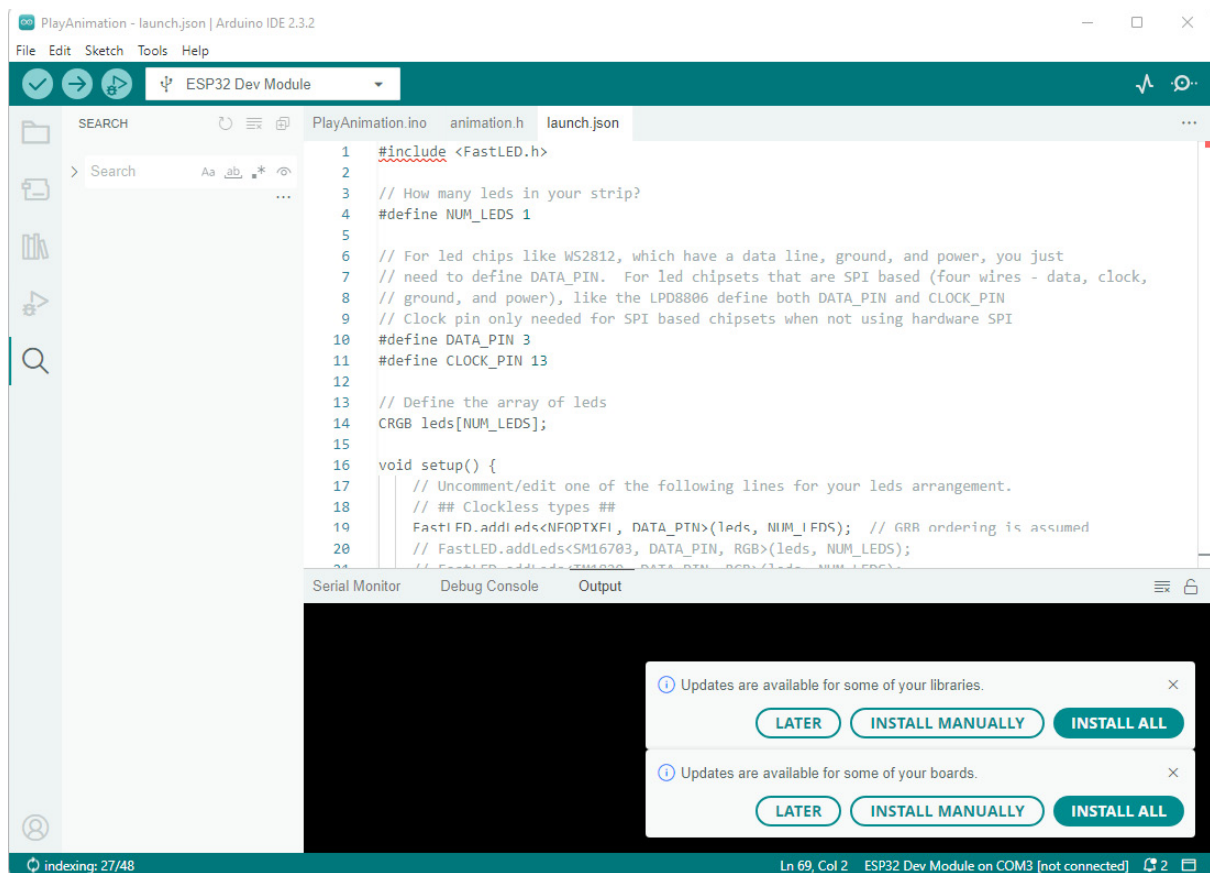
Jak już wspomniałem, środowisk i języków programowania dostępnych dla układów Espressif jest sporo i na pewno każdy znajdzie coś dla siebie. Sam preferuję czysty język C i oficjalne SDK ESP-IDF. Niemniej nie jest moim celem namawianie w tej książce kogokolwiek do czegokolwiek i udowadnianie, że dane środowisko jest jedynym słusznym. Dlatego też tutaj skupimy się na dwóch popularnych środowiskach: Arduino i PlatformIO. (Może i słowo "środowisko" jest tu użyte trochę na wyrost, ale niech tak zostanie). Każde z nich ma swoje atuty, przy czym drugie całkowicie zastępuje to pierwsze, zapewniając przy tym znacznie więcej możliwości.

Gdy mówimy o środowisku programistycznym, zazwyczaj mamy na myśli IDE oraz zestaw frameworków, kompilatorów i bibliotek, które udostępnia producent sprzętu lub skupiona wokół niego społeczność. IDE powinno nam zaoferować wygodny edytor kodu z kolorowaniem składni i podpowiedziami (IntelliSense) oraz szybki dostęp do narzędzi, zarówno sprzętowych, ułatwiających wgrywanie oprogramowania, jak też opcji debugera, służących do wyłapywania błędów i śledzenia wykonywania programu. Oczywiście nie zawsze jest tak kolorowo — niektóre środowiska to opasłe "kombajny" przytłaczające ogromem funkcji, inne natomiast są proste i udostępniają minimum możliwości — ale jak mówiłem, co kto lubi i gdzie się czuje wygodniej.

6. Arduino IDE

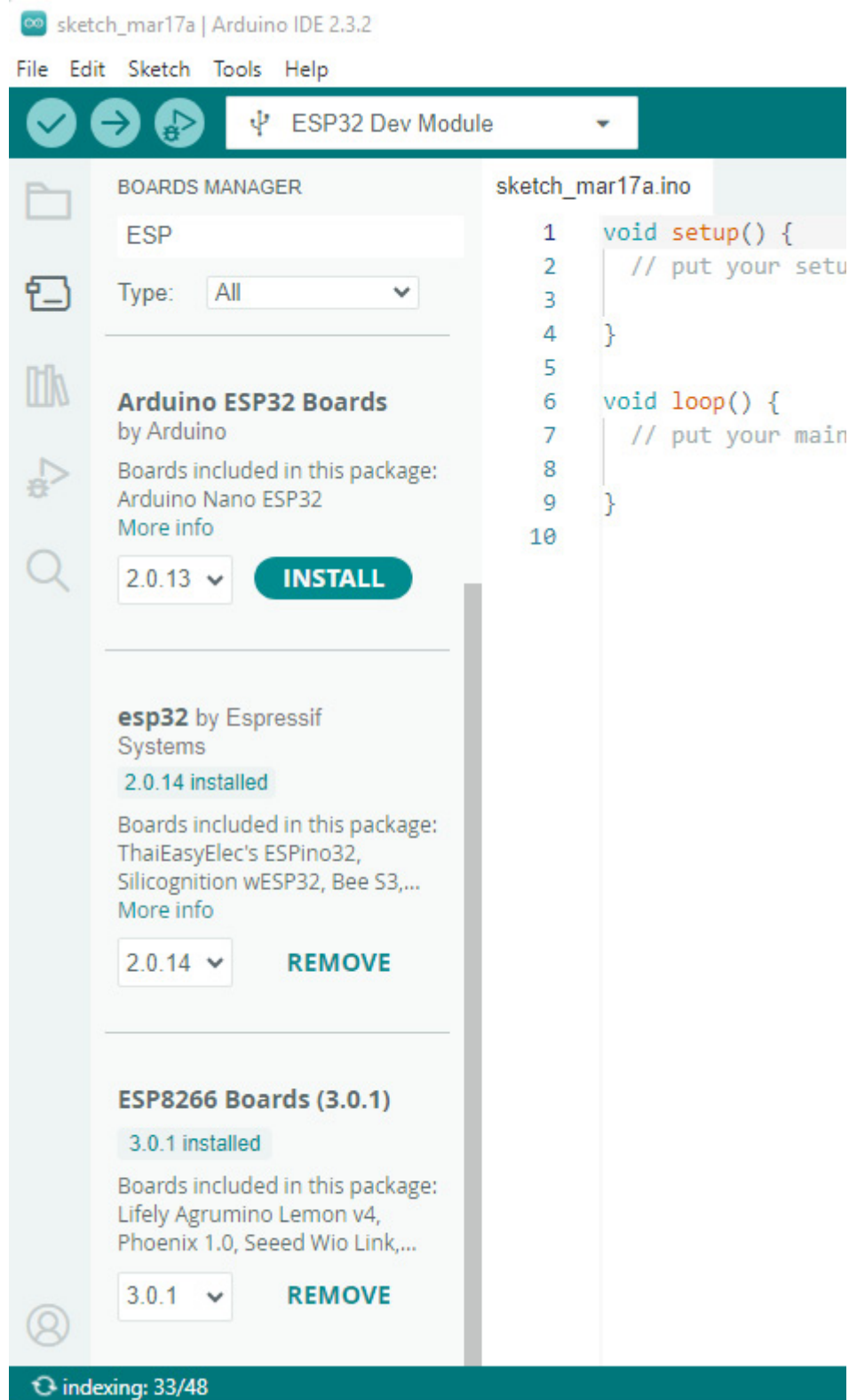
Arduino IDE jest właśnie takim najprostszym, minimalistycznym środowiskiem, które umożliwia pisanie kodu, ma kolorowanie składni i umożliwia wygodne manipulowanie płytkami i bibliotekami. Nie jest idealne, ale ma sporą społeczność, dzięki czemu udostępnia dobrze zrobione porty dla różnych mikrokontrolerów wraz z bogatymi przykładami i bibliotekami. Oczywiście piszemy tu kod w czymś, co z grubsza można nazwać językiem C++, który ja postrzegam jako, powiedzmy, etap ewolucji Basica. Prosta składnia i specyficzne rozkazy dla mikrokontrolerów mogą zdecydowanie ułatwić początki w programowaniu, jak też być dla niektórych dobrą alternatywą dla zawłości czystego języka C. Ta właśnie otoczka tworzy całkiem przyjemne i przystępne środowisko dla każdego, a bogactwo bibliotek i przykładów pozwala łatwo tworzyć wielkie rzeczy.

By wygodnie pracować w tym środowisku, nie trzeba mieć wiedzy tajemnej, a całość ogranicza się do pobrania Arduino IDE ze strony producenta, dostępnej pod adresem <https://www.arduino.cc/>. Samo środowisko przez lata mocno się zmieniło — mamy już choćby edytor v2 i doszła chmura dla urządzeń IoT — i można powiedzieć, że od czasów małego projektu, jakim było na początku, zrobiło się z niego dość poważne narzędzie (rysunek 6.1).



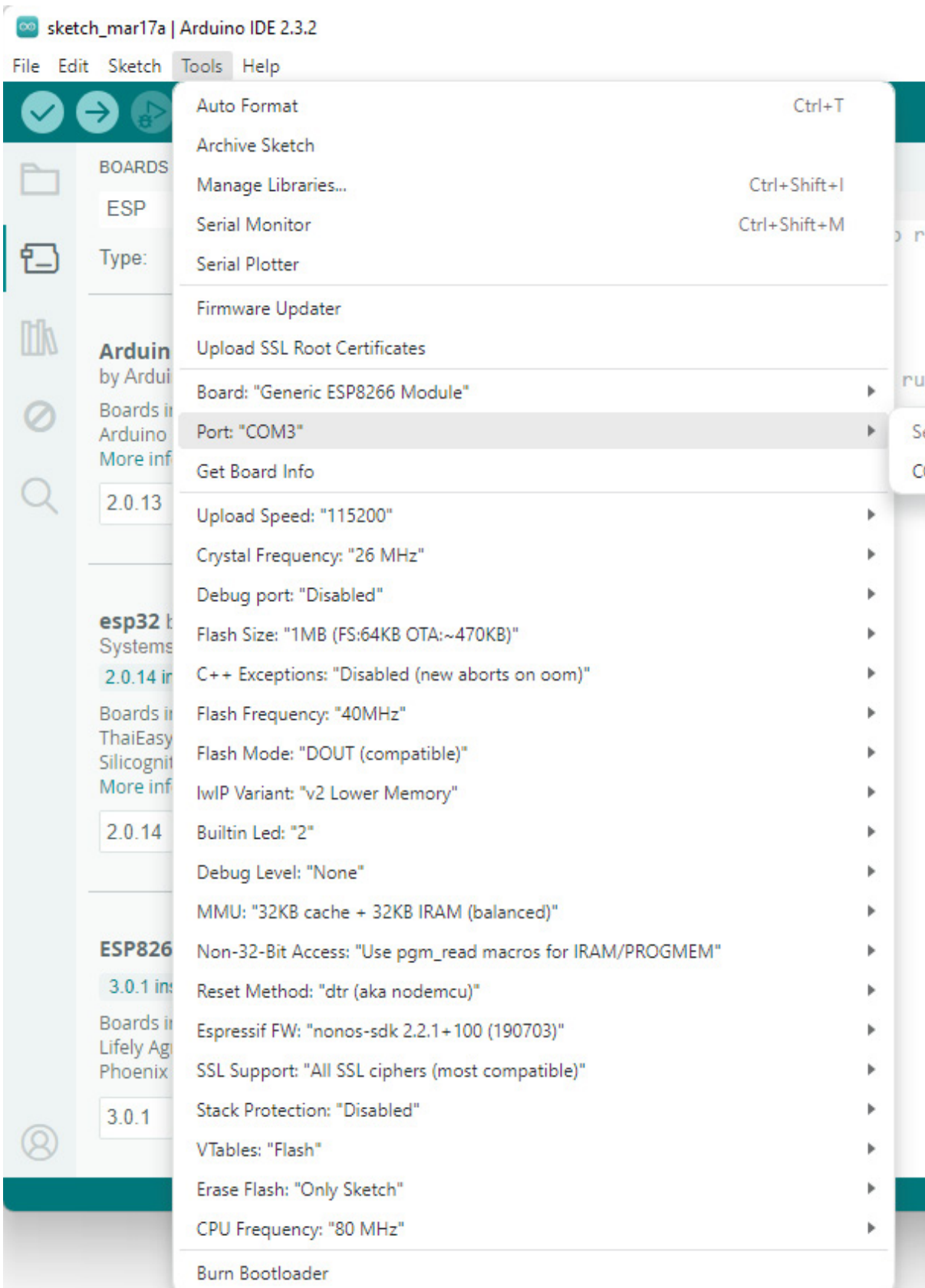
Rysunek 6.1. Okno środowiska Arduino

Po zainstalowaniu środowiska należy zainstalować obsługę wybranych układów i dostępnych płytek, a także framework z kompilatorem, opisem portów i bibliotekami oraz podstawową listą przykładów dla naszego ESP. Tak więc po zainstalowaniu środowiska i jego uruchomieniu wybieramy *BOARDS MANAGER* (rysunek 6.2).



Rysunek 6.2. Boards Manager w Arduino IDE

Wpisujemy **ESP** i instalujemy frameworki dla ESP8266 i ESP32. Chwilę to potrwa, mniej lub więcej w zależności od szybkości komputera i łącza internetowego, więc warto się uzbroić w kawę i nieco cierpliwości. Po tym środowisko będzie gotowe do użycia, co możemy łatwo sprawdzić: wystarczy wybrać jakiś przykładowy program i wgrać go na płytkę. Wybierzmy więc np. Generic ESP8266 Module, który pasuje do większości dostępnych płytek z układem ESP8266 (rysunek 6.3).



Rysunek 6.3. Menu Tools w Arduino IDE

Na razie nie musimy nic zmieniać w ustawieniach płytki z wyjątkiem portu *COM*, na którym płytka jest podłączona, by umożliwić sobie wgrywanie wsadu do pamięci flash. Tę zabawę pozostawiam Wam, żebyście mogli sprawdzić działanie środowiska. Opiszę tylko pokrótce strukturę programu na przykładzie prostego przykładowego programu Blink.

Omówienie struktury programu w Arduino

Gdy wybierzemy z menu *File/Examples/ESP8266/Blink*, zostanie załadowany przykładowy program, który nie robi nic specjalnego, a mianowicie miga diodą LED dołączoną do pinu GPIO. To swoisty "Hello World" w świecie mikrokontrolerów, pokazujący pierwszą i najprostszą interakcję mikrokontrolera z użytkownikiem. Arduino ma dość specyficzną strukturę programu, którą dobrze widać właśnie na przykładzie programu Blink. Ze względu na prostotę programu łatwo będzie nam omówić jego działanie. Zerknijmy więc na jego kod:

```
/*  
    ESP8266 Blink by Simon Peter  
  
*/  
  
void setup()  
{  
    pinMode(LED_BUILTIN, OUTPUT);
```

```
}

void loop()
{
    digitalWrite(LED_BUILTIN, LOW);
    delay(1000);
    digitalWrite(LED_BUILTIN, HIGH);
    delay(1000);
}
```

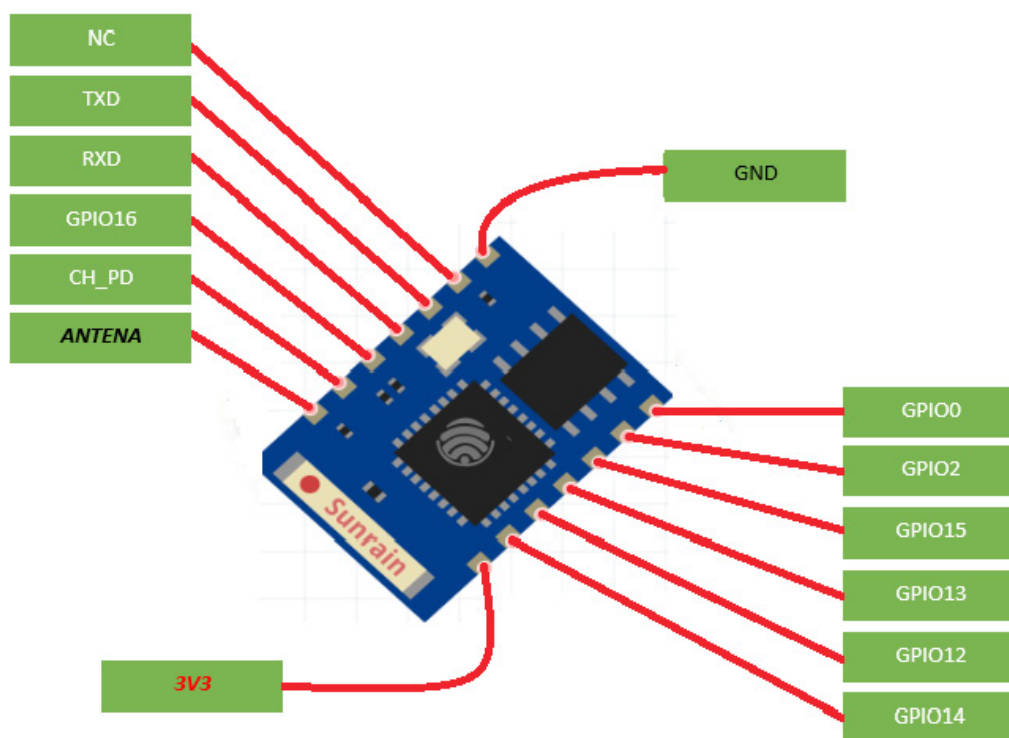
Jak widzicie, całość jest wręcz banalna. Oczywiście oryginalnie była opatrzona sporą liczbą komentarzy objaśniających, które usunąłem, by nie zaciemniać struktury programu. W Arduino pierwszą częścią programu jest funkcja `setup()`. W niej umieszczamy konfigurację pinów GPIO, peryferiów i różne inne rzeczy, które podczas uruchomienia naszego programu powinny wykonać się tylko raz. Tu za wiele nie ma:

```
void setup()
{
    pinMode(LED_BUILTIN, OUTPUT);
}
```

Właściwie w tym przypadku ustawiamy tylko pin nazwany *LED_BUILTIN*, który odnosi się do podłączonej fizycznie diody LED na płytce — jako wyjście.

	Przykład dotyczy modułu ESP-01, więc jeśli macie inną płytkę niż ta, dioda Wam nie zamiga i będziecie przekonani, że program albo płytką nie działa. Dlatego też warto zapoznać się z dokumentacją swojej płytki i zobaczyć, który pin ma podpiętą jakąś diodę. Moja, Rounded Board, jest oparta na module ESP-03, zatem w odróżnieniu od ESP-01 ma siedem GPIO, za to nie ma żadnej własnej diody. Jednak na płytce umieściłem dwie, pokazujące działanie przekaźników, więc możemy ich użyć, co poza efektem wizualnym spowoduje efekt dźwiękowy w postaci klikania przekaźnika.
---	---

Na początek zobaczmy pinout ESP-03, pokazany na rysunku 6.4.



Rysunek 6.4. Pinout modułu ESP-03¹

U mnie jeden przekaźnik jest podpięty do pinu *GPIO14*, a drugi do *GPIO12*. Musimy więc zmienić konfigurację pinu w

programie.

```
pinMode(LED_BUILTIN, OUTPUT);
```

zmieniamy na:

```
pinMode(12, OUTPUT);
```

I to wszystko w tym miejscu. Właśnie wskazaliśmy kompilatorowi, że naszym pinem wyjściem jest *GPIO12*. Dzieje się tak dlatego, że instrukcja `pinMode()` przyjmuje dwa parametry: numer pinu i tryb pracy. Czyli wygląda to tak:

```
pinMode(<pin>, <tryb pracy>);
```

gdzie *pin* to po prostu numer pinu lub nazwa pinu (wyjaśnię to później), a *tryb pracy* to jeden z trybów, w jakich może pracować pin, czyli:

- OUTPUT — wyjście,
- INPUT — wejście,
- INPUT-PULLUP — wejście z rezystorem podciągającym.

W tej chwili trzeci tryb pracy nie jest dla nas istotny i zasadniczo może wcale się nam nie przydać, niemniej istnieje, a później się wyjaśni, z czym to się je.

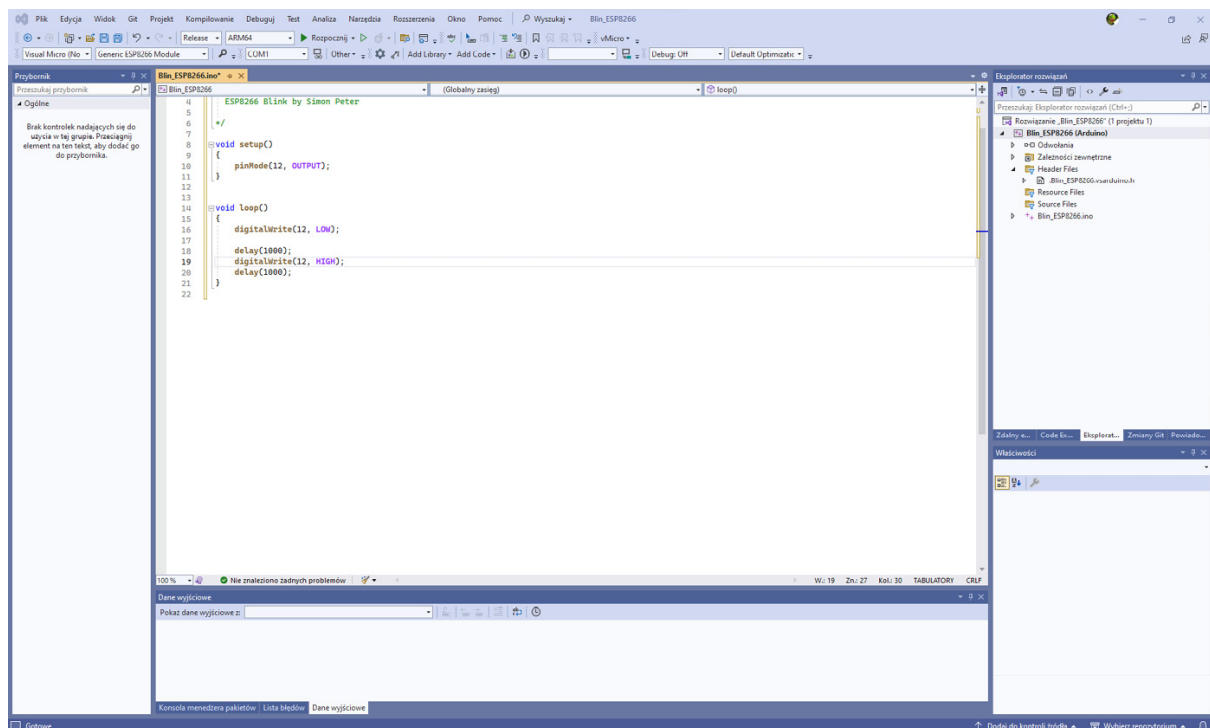
Gdy już wyjaśniliśmy ideę funkcji `setup` i ustawiliśmy pin w naszym ESP8266, możemy zająć się drugą częścią programu, czyli funkcją `loop()`. To jego główna funkcja. Zawarte w niej instrukcje są wykonywane w pętli przez cały czas działania programu. W naszym przykładowym programie w funkcji `loop()` zawarte są tylko dwie instrukcje:

```
void loop()
{
    digitalWrite(12, LOW);
    delay(1000);
```

```
digitalWrite(12, HIGH);  
delay(1000);  
}
```

Instrukcja `digitalWrite()` ustawia stan niski na wybrany pin, następnie instrukcja `delay()` powoduje opóźnienie o zadany czas, potem kolejne wywołanie instrukcji ustawia na pinie stan wysoki i znów czekamy, i tak w kółko. Jak prawdopodobnie zauważyliście, tu też musimy zmienić LED-BUILTIN na nasz numer pinu. Zauważyliście też pewnie, że instrukcja `digitalWrite()` przyjmuje wartości: pin, stan, gdzie stan to LOW lub HIGH. Instrukcja `delay()` zawiera zaś wartość opóźnienia w milisekundach. W tym przypadku $1000 = 1 \text{ s}$, tak więc nasza dioda będzie zapalać się i gasnąć co 1 sekundę. Widzimy, że instrukcje mają proste nazwy, jednoznacznie kojarzące się z funkcją, jaką wykonują. I to właśnie stanowi o prostocie Arduino i zapewnia, że jest przystępne dla wielu ludzi nie mających wiele wspólnego z programowaniem, w myśl zasady: programować każdy może.

No dobrze, to tylko podstawy podstaw, jednak w tej chwili nie będziemy bardziej się zagłębiać w Arduino, gdyż opisów w internecie nie brakuje, a język, jak widzicie, ma prostą i przejrzystą składnię. Gorzej z samym środowiskiem i edytorem. Ten wprawdzie jest prosty w użyciu, ale uważam, że nie jest zbyt poręczny i niektóre rzeczy wręcz utrudnia, na przykład gdy piszemy duży kod zawierający wiele bibliotek, w których musimy również wprowadzać zmiany, lub piszemy własne biblioteki i rośnie nam liczba plików w projekcie. Arduino utrudnia nam zarządzanie nimi i dostęp do nich, zwłaszcza do tych wbudowanych podstawowych. Dlatego warto skorzystać z innych rozwiązań, takich jak plugin VisualMicro, czyli Arduino dla Visual Studio Community, pokazane na rysunku 6.5.

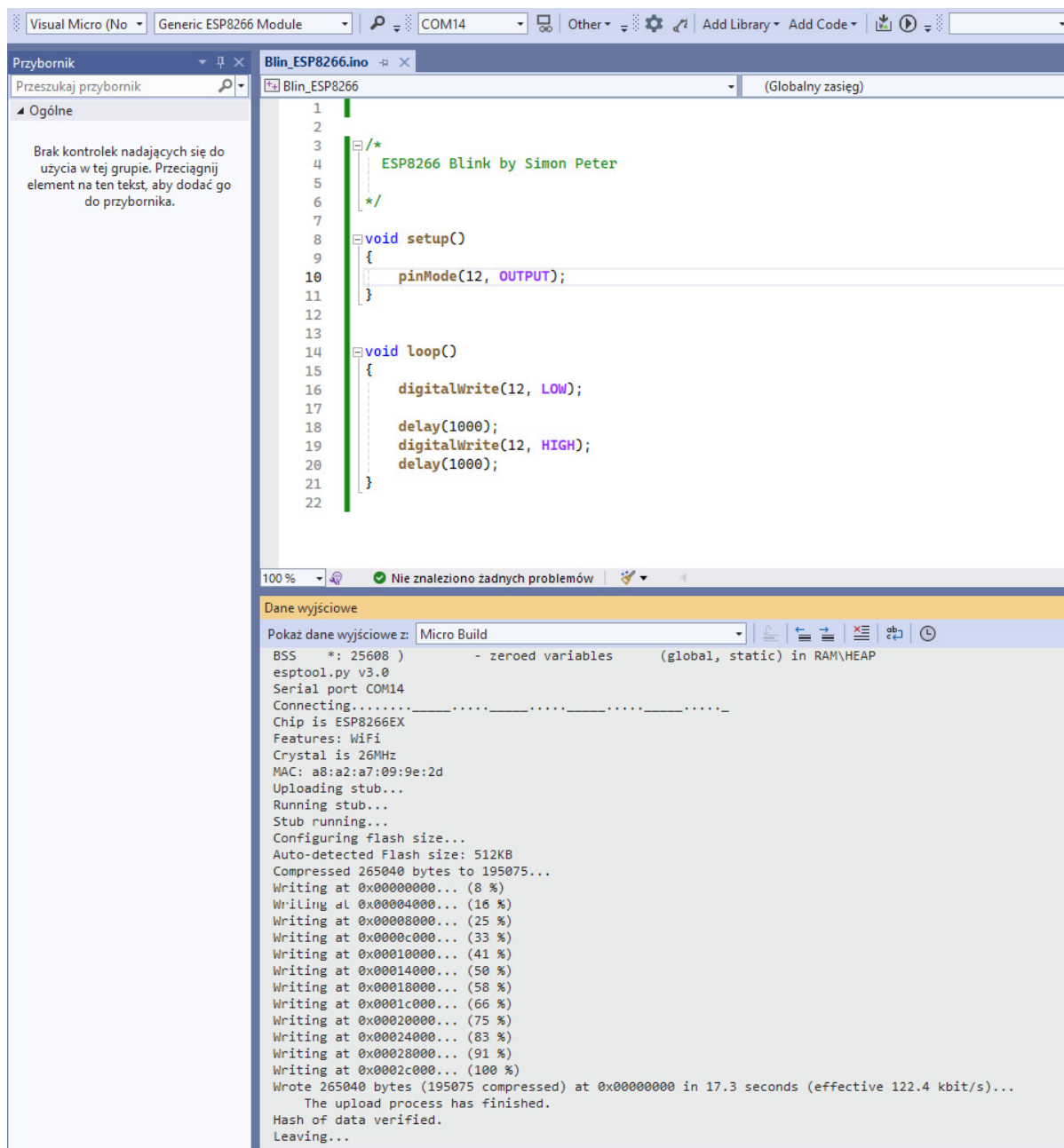


Rysunek 6.5. Visual Studio Community 2022

Tu zarządzanie projektem jest bardzo wygodne. Niestety plugin jest płatny. Wiele jednak daje nawet w tak dużym środowisku, jakim jest Visual Studio Community, dlatego warto się nim zainteresować i choćby wypróbować, dostajemy bowiem 45-dniowy darmowy okres próbny, po którego upływie możemy zdecydować, czy kupujemy roczną licencję, czy też nie. Nie namawiam nikogo do jego zakupu, choć nie ukrywam, że dla mnie jest bardzo wygodne, gdyż dużo programuję w Visual Studio i wolę mieć wszystko w jednym i wygodnym IDE.

Wróćmy teraz do naszego programu i zobaczmy, czy działa.

Program skompilował się poprawnie, bez błędów, został wgrany za pomocą bootloadera do płytki, zielona dioda radośnie miga, a przekaźnik klika. Przebieg kompilacji programu i wgrywania wsadu do pamięci flash jest pokazany na rysunku 6.6.



Rysunek 6.6. Przebieg kompilacji programu i wgrywania wsadu do pamięci flash w środowisku

Gwoli wyjaśnienia: moja płytką nie ma wbudowanego układu komunikacyjnego, takiego jak Silabs CP2102 czy CH340, dlatego w tym miejscu posiłkuję się właśnie przedstawionym już ESP-PROG-iem, czyli programatorem-

debugerem przeznaczonym specjalnie do układów ESP firmy Espressif, ale może to być dowolny tani układ konwertera USB-UART. Oczywiście o ile płytka nie ma swojego gniazda USB z układem komunikacyjnym.





Rysunek 6.7. Programator ESP-PROG z płytką CircleBoard

Na zdjęciu pokazanym na rysunku 6.7 zielona dioda symbolizuje właśnie nasz pin 12. Szkoda, że tego nie słyszycie, ale radośnie klika również przekaźnik. Tak więc możemy uznać, że framework Arduino IDE działa poprawnie i możemy go używać do pisania programów dla naszego ESP.

1 Źródło: Schemat stworzony w narzędziu Fritzing; dostęp 11.06.2024.

7. Visual Studio Code i PlatformIO

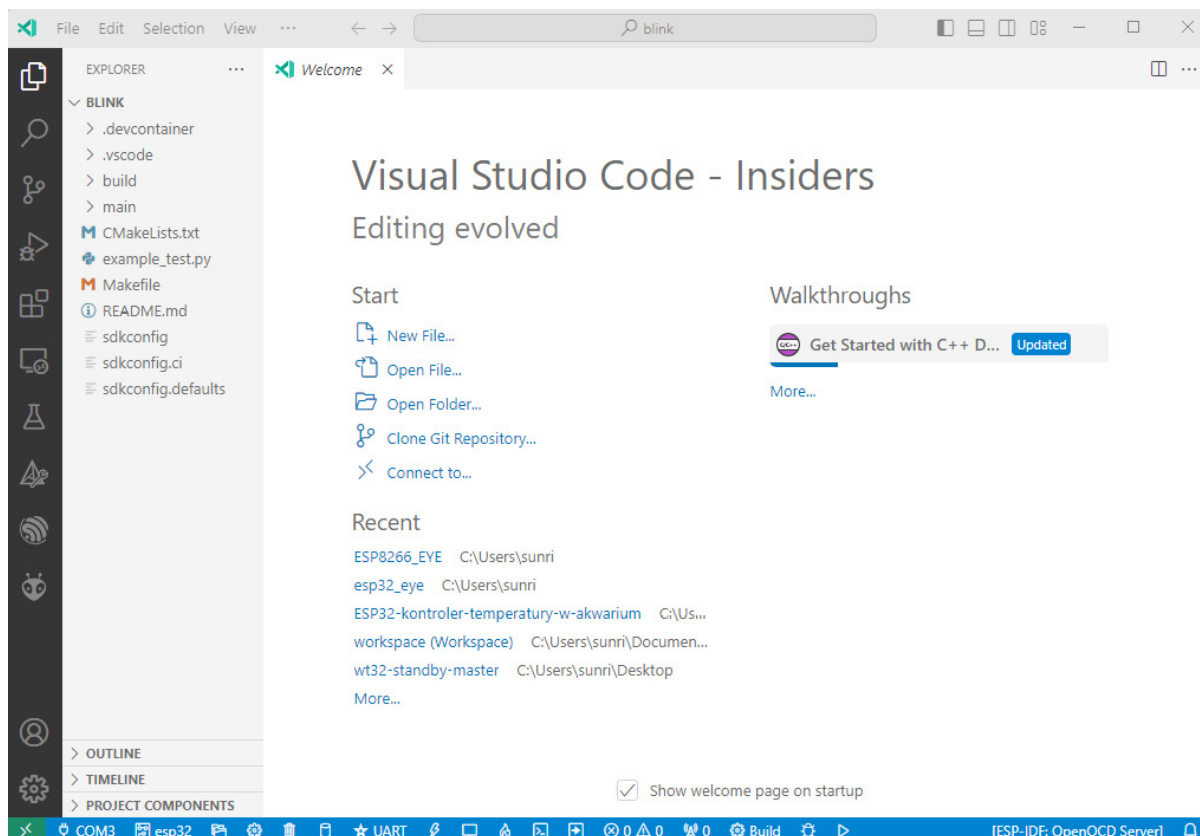
Visual Studio Code to zaawansowane, ale lekkie środowisko IDE, zwłaszcza w porównaniu z "dużym" Visual Studio. Visual Studio Code jest bezpłatne, tak samo jak Visual Studio Community. Microsoft, choć nieczęsto mu się to zdarza, tu naprawdę zrobił kawał dobrej roboty. IDE jest małe i lekkie, a ponadto wieloplatformowe i pozwala łatwo integrować różne frameworki, od typowych języków programowania, czyli C, C++, F#, C#, Python, Java, do tworzenia aplikacji desktopowych i konsolowych, po frameworki dla *embedded*. Jako edytor kodu ma wiele zalet i jest bardzo wygodny, a jako środowisko pozwala przekraczać bariery. I choć wygląd ma minimalistyczny, nie sposób go nie docenić.

Visual Studio Code pobieramy ze strony <https://code.visualstudio.com/>. Dostępne są wersje dla następujących systemów:

- Windows,
- Linux,
- macOS.

Taki wybór pozwala każdemu użyć tego środowiska oraz pracować wydajnie i wygodnie w wielu językach programowania. Narzędzie umożliwia zarówno pisanie kodu, jak i debugowanie i uruchamianie programów. Zapewnia też pełną funkcję IntelliSense, integrację z systemem kontroli wersji Git z wykorzystaniem witryny GitHub oraz wiele dostępnych pluginów. I to właśnie wybranym pluginom teraz się przyjrzymy.

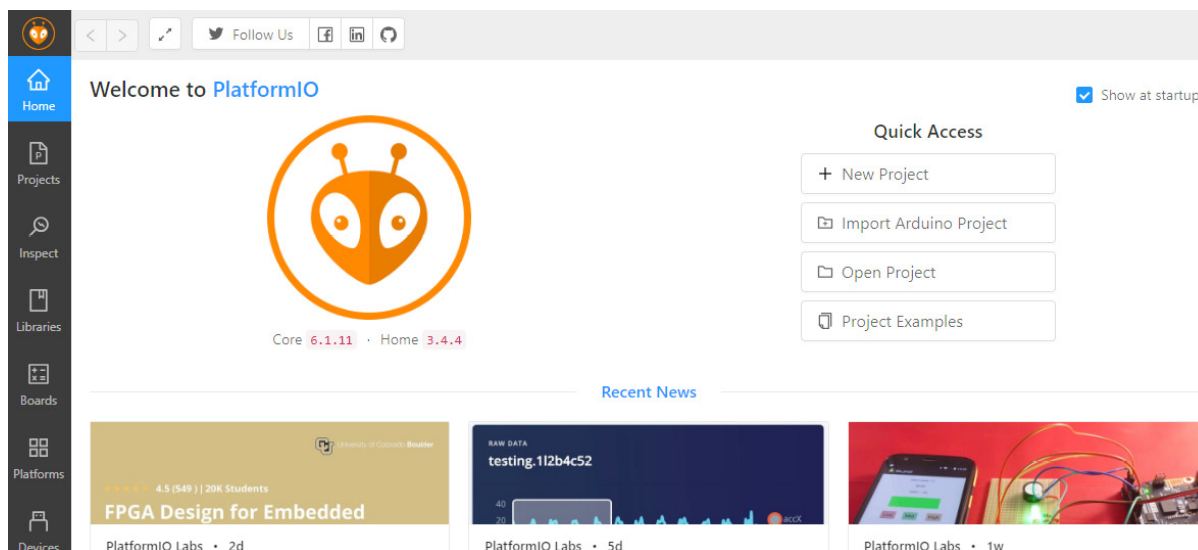
Po zainstalowaniu Visual Studio Code, pokazanego na rysunku 7.1, możemy wybrać, co nam jest potrzebne, poprzez menedżera rozszerzeń. Jednym z istotnych jest PlatformIO. W tej chwili to bardzo potężne rozszerzenie, które po integracji z VSC udostępnia ponad 40 platform sprzętowych, ponad 20 frameworków, aż 1,5 tys. płytek i zestawów developerskich oraz ponad 13 tys. bibliotek. To dużo, bardzo dużo, a jeszcze nie wszystko. PlatformIO w pełni zastępuje Arduino IDE, pozwalając na wygodną pracę z frameworkiem Arduino i wszelkimi jego zasobami, również pod kontrolą oficjalnego SDK Espressif.



Rysunek 7.1. Widok okna środowiska Visual Studio Code

Opis instalacji PlatformIO znajdziecie na stronie <https://docs.platformio.org/en/latest/integration/ide/vscode>.

html#installation. Jej przebieg jest banalnie prosty. Instalujemy VSC, otwieramy menedżera rozszerzeń, wpisujemy **platformio** i klikamy *Install*.



Rysunek 7.2. Plugin PlatformIO zainstalowany w VSC

I gotowe (rysunek 7.2). Podobnie też z poziomu PlatformIO dodajemy frameworki, które są nam potrzebne.

Tworzymy projekt

Gdy już wszystko mamy zainstalowane, użyjemy kreatora projektu (*Project Wizard*) do utworzenia nowego projektu. Nazwiemy go *TestESP* i wybierzemy płytkę Espressif Generic ESP8266 ESP-01 1M. (Opcji z *Generic* w nazwie używamy zawsze, gdy konkretnego modelu płytki brakuje na liście obsługiwanych konstrukcji). Następnie wybieramy framework (rysunek 7.3).

Project Wizard

This wizard allows you to **create new** PlatformIO project or **update existing**. In the last case, you need to uncheck "Use default location" and specify path to existing project.

Name: TestESP

Board: Espressif Generic ESP8266 ESP-01 1M

Framework: Arduino

Location: Arduino
ESP8266 RTOS SDK
ESP8266 Non-OS SDK

Cancel Finish

Rysunek 7.3. PlatformIO — okno kreatora projektu (Project Wizard)

Jak widzicie, w przypadku ESP8266 możemy wybrać framework Arduino albo ESP8266 RTOS SDK lub ESP8266 Non-OS SDK. Pierwszy to po prostu implementacja Arduino, drugi to oficjalne SDK działające pod kontrolą RTOS (ang. *Real Time Operation System*), trzeci zaś to czyste SDK. Na razie sprawdzimy na szybko framework Arduino z naszym programikiem Blink.

Tu zapewne nastąpi małe zdziwienie, gdyż po utworzeniu projektu zobaczycie inny szablon struktury programu:

```
#include <Arduino.h>

// put function declarations here:

int myFunction(int, int);
```

```

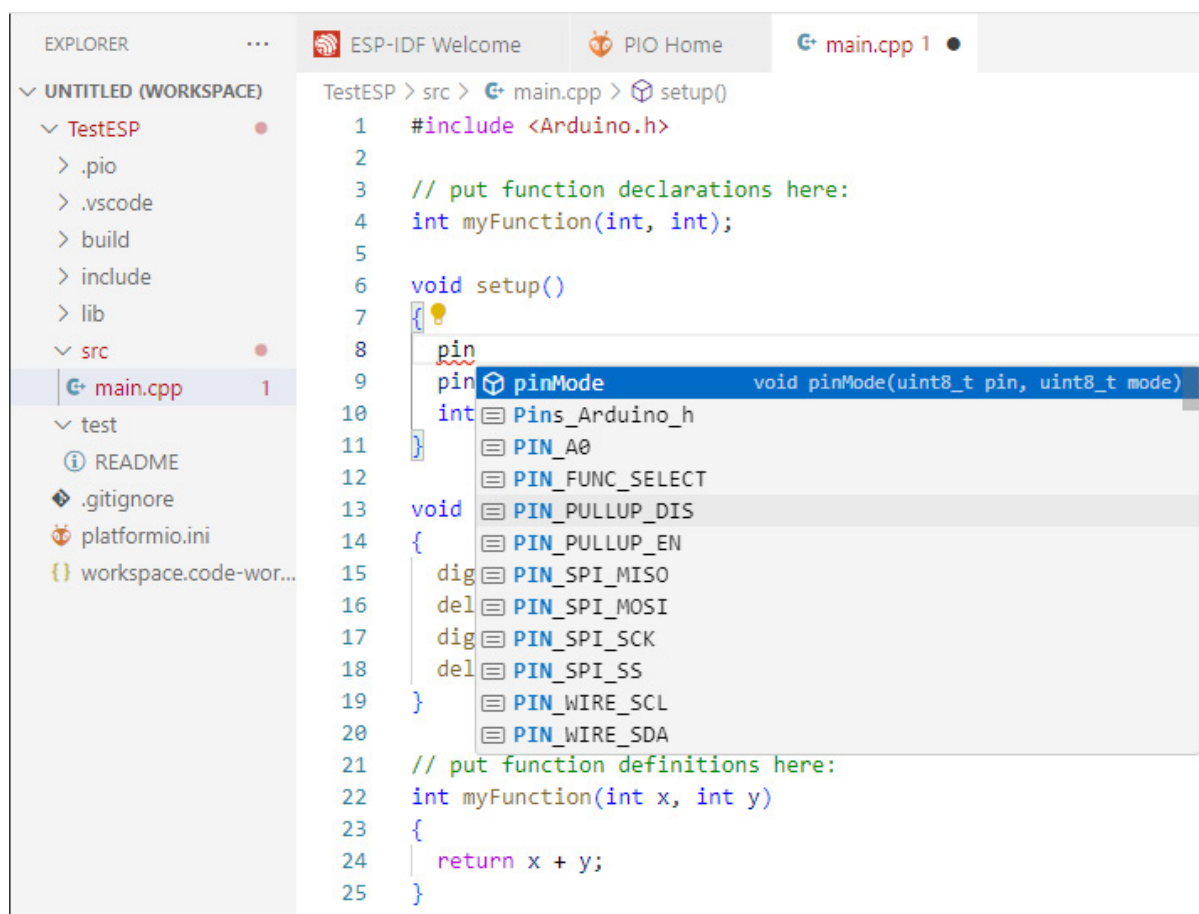
void setup() {
    // put your setup code here, to run once:
    int result = myFunction(2, 3);
}
void loop() {
    // put your main code here, to run repeatedly:
}
// put function definitions here:
int myFunction(int x, int y) {
    return x + y;
}

```

Niemniej tak naprawdę nic się nie zmieniło. Oczywiście w pierwszej linijce widzimy instrukcję `#include` pliku nagłówkowego *Arduino.h*, który w przypadku PlatformIO jest istotny, gdyż pokazuje, z jakiego frameworka będziemy korzystać, i zawiera opisy wymagane przez Arduino. Po tej instrukcji następuje deklaracja funkcji `int myFunction(int, int)`, która pokazuje, że w tym miejscu kodu będziemy deklarować nasze funkcje, choćby właśnie `int`, która przyjmuje wartości `int` i takie też zwraca (funkcje `void` nie zwracają żadnych wartości).

Następnie mamy już tradycyjnie `void setup()` i `void loop()`, które nie zwracają żadnych wyników, oraz, na końcu, naszą funkcję `int myFunction()`, która przyjmuje wartości `int` dla `x` i `y`, a następnie zwraca wynik dodania tych wartości. Obecnie to nam się nie przyda, ale zauważcie, że funkcja wykona się wewnątrz funkcji `setup`, czyli tylko raz, po uruchomieniu programu. Jak więc widzicie, prosta struktura programu Arduino nieco się rozszerzyła o deklaracje funkcji i ciało funkcji. A z tego już niekiedy będziemy korzystać. Zaczynamy zatem w sekcji `setup` wpisywać przedstawioną już instrukcję `pinMode()` i tu

właśnie widać, jak fajnie działa podpowiadanie składni (rysunek 7.4). Po prostu aż miło się pracuje.



Rysunek 7.4. Podpowiadanie składni poleceń

To ważne ułatwienie, zwłaszcza gdy piszemy obszerny program i trudno nam spać pamiętać wszystkie zmienne, stałe i nazwy funkcji. Tego w samym Arduino nie znajdziecie.

Jak widać, nasze ulepszone środowisko działa poprawnie również w połączeniu z frameworkiem Arduino (rysunek 7.5).

```
TestESP > src > main.cpp > loop()
1  #include <Arduino.h>
2
3  // put function declarations here:
4  int myFunction(int, int);
5
6  void setup()
7  {

* Executing task in folder TestESP: C:\Users\sunri\.platformio\penv\Scripts\platformio.exe run
Compiling .pio\build\esp01_1m\FrameworkArduino\spiffs\spiffs_nucleus.cpp.o
Compiling .pio\build\esp01_1m\FrameworkArduino\spiffs_api.cpp.o
Compiling .pio\build\esp01_1m\FrameworkArduino\sqr32.cpp.o
Compiling .pio\build\esp01_1m\FrameworkArduino\stdlib_noniso.cpp.o
Compiling .pio\build\esp01_1m\FrameworkArduino\time.cpp.o
Compiling .pio\build\esp01_1m\FrameworkArduino\uart.cpp.o
Compiling .pio\build\esp01_1m\FrameworkArduino\umm_malloc\umm_info.c.o
Compiling .pio\build\esp01_1m\FrameworkArduino\umm_malloc\umm_integrity.c.o
Compiling .pio\build\esp01_1m\FrameworkArduino\umm_malloc\umm_local.c.o
Compiling .pio\build\esp01_1m\FrameworkArduino\umm_malloc\umm_malloc.cpp.o
Compiling .pio\build\esp01_1m\FrameworkArduino\umm_malloc\umm_poison.c.o
Compiling .pio\build\esp01_1m\FrameworkArduino\wpa2_eap_patch.cpp.o
Archiving .pio\build\esp01_1m\libFrameworkArduino.a
Linking .pio\build\esp01_1m\firmware.elf
Retrieving maximum program size .pio\build\esp01_1m\firmware.elf
Checking size .pio\build\esp01_1m\firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM: [====] 34.3% (used 28104 bytes from 81920 bytes)
Flash: [====] 34.3% (used 261479 bytes from 761840 bytes)
Building .pio\build\esp01_1m\firmware.bin
Creating BIN file ".pio\build\esp01_1m\firmware.bin" using "C:\Users\sunri\.platformio\packages\framework-arduinoespressif8266\bootloaders\boot\boot.elf" and ".pio\build\esp01_1m\firmware.elf"
===== [SUCCESS] Took 14.61 second
=====
* Terminal will be reused by tasks, press any key to close it.
```

Rysunek 7.5. Podgląd kompilacji i wgrywania zawartości do pamięci flash

Program jest skompilowany i poprawnie wgrany. Zmieniłem przy tym wartość `delay(1000)` na `delay(500)`, co powoduje, że nasz przekaźnik klika co 0,5 s i tak samo często miga dioda LED.

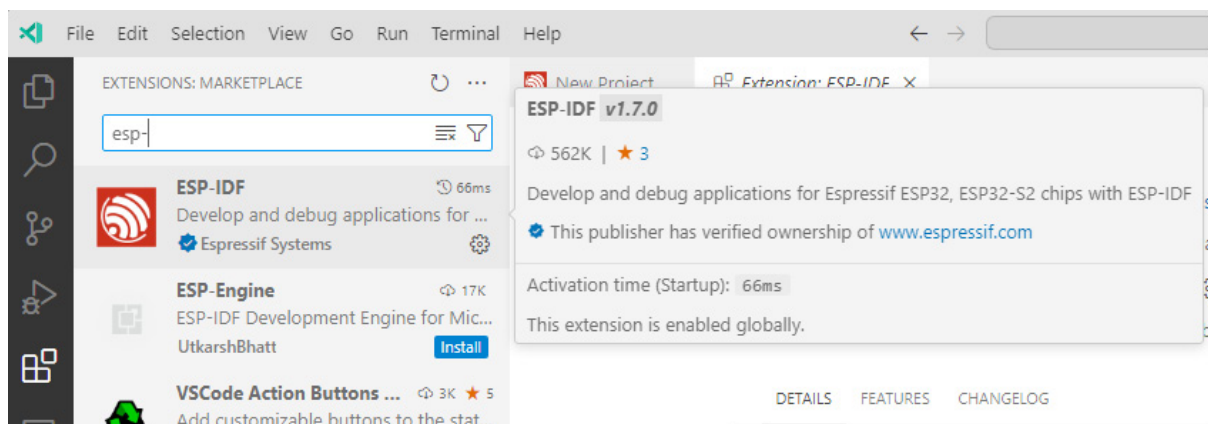
Możemy też wykonać ten sam test dla płytki ESP32 One, która ma na pokładzie dwurdzeniowy układ ESP32 oraz własny układ komunikacyjny, Silabs.

8. Visual Studio Code — ESP-IDF

Tu, jak już wspomniałem, PlatformIO pozwala użyć również SDK Espressif, czyli IDF, ale chcę Wam jeszcze w tym miejscu pokazać alternatywne rozwiązanie. Dlatego też zintegrujemy Visual Studio Code z SDK ESP IDF firmy Espressif bezpośrednio, bez użycia pluginu PlatformIO. W tym celu najpierw musimy zainstalować pakiet SDK wraz z narzędziami i zapisać jego ścieżki. Instalator pobieramy ze strony <https://dl.espressif.com/dl/esp-idf/>.

Pakiet instalujemy w ścieżkach, które proponuje nam instalator. Wraz z pakietem zainstaluje się też Python, Git i potrzebne narzędzia. Teraz możemy uruchomić Visual Studio Code.

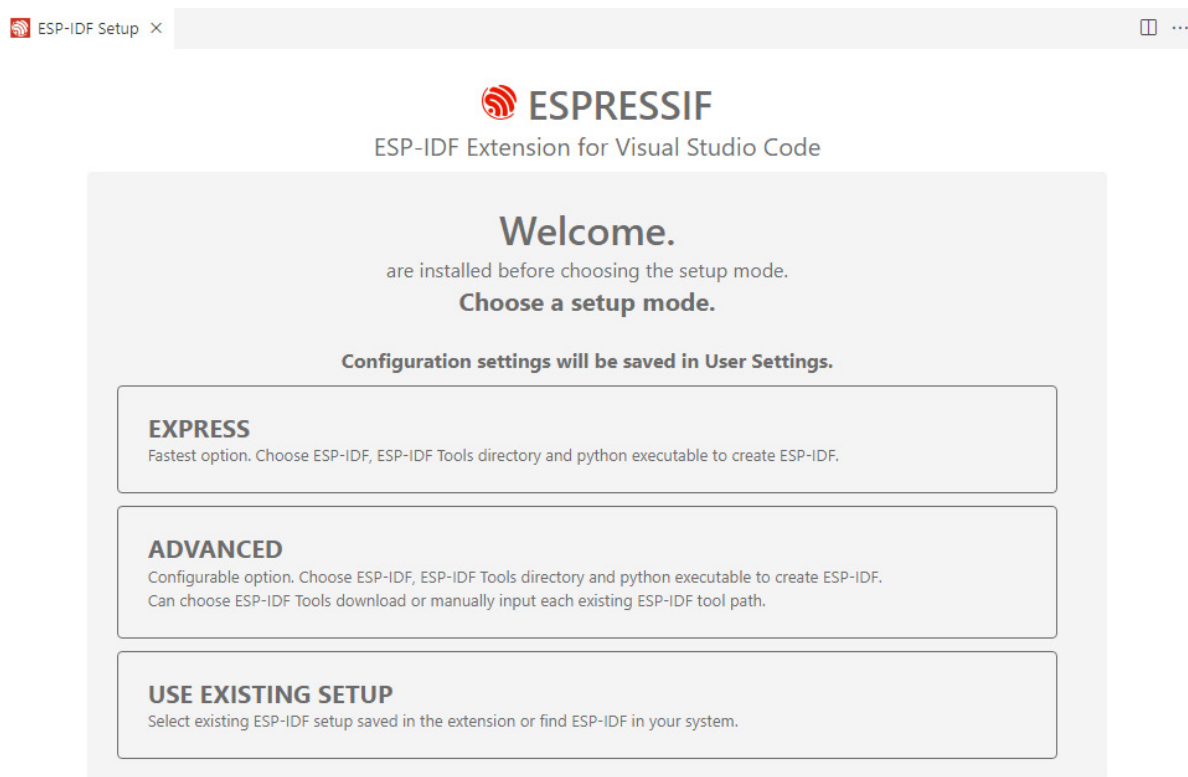
W nim naciskamy *Ctrl+P* i wyszukujemy rozszerzenie *ESP-IDF* (rysunek 8.1).



Rysunek 8.1. ESP-IDF w Marketplace VSC

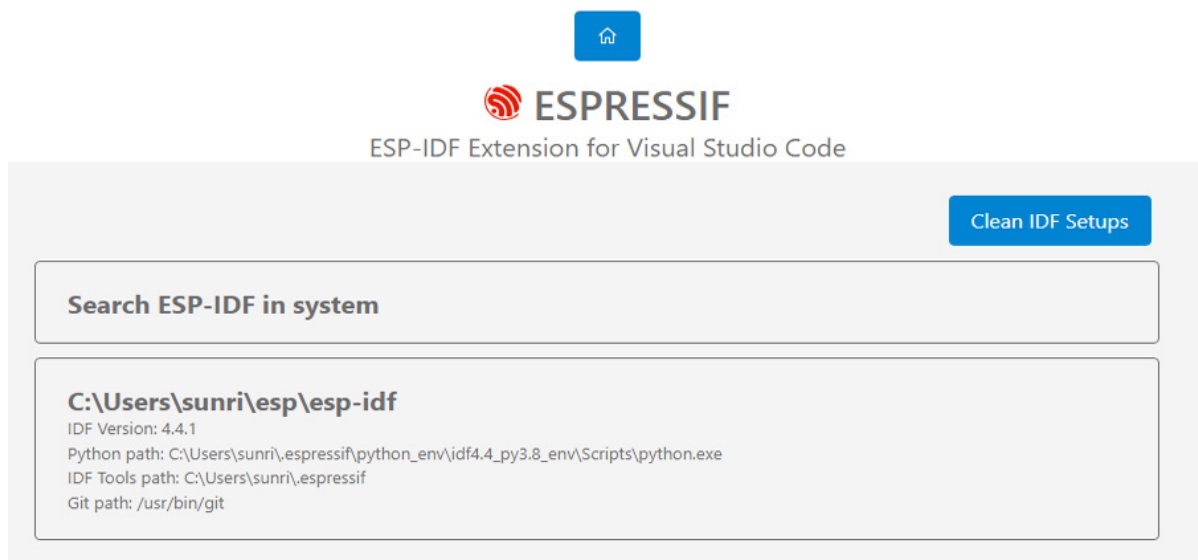
Klikamy *Install* (u mnie już jest zainstalowane). Po zainstalowaniu naciskamy *F1* i wybieramy *Configure ESP-*

IDF extension. Po chwili otworzy nam się strona konfiguracji (rysunek 8.2).



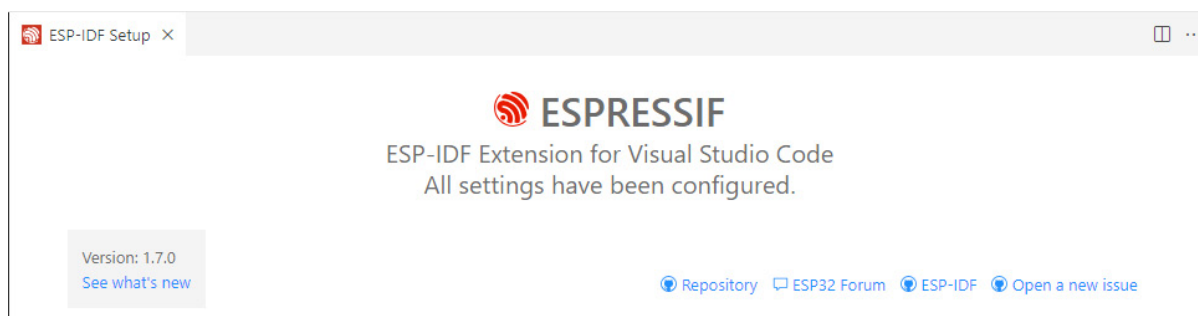
Rysunek 8.2. Ekran konfiguracji ESP-IDF — wybór wariantów konfiguracji

Wybieramy *USE EXISTING SETUP*. Ścieżka zostanie automatycznie odnaleziona (rysunek 8.3).



Rysunek 8.3. Ekran konfiguracji ESP-IDF — wybór ścieżki

Klikamy ją, i po chwili środowisko zostanie skonfigurowane (rysunek 8.4).

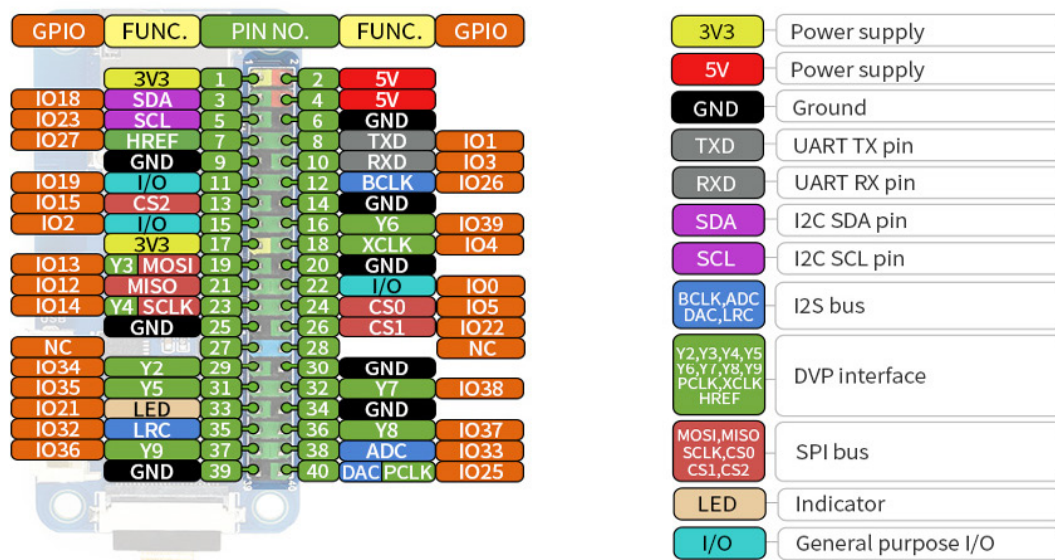


Rysunek 8.4. Ekran konfiguracji ESP-IDF — podsumowanie

Pierwszy program w ESP-IDF

Teraz możemy przystąpić do utworzenia w naszym środowisku nowego projektu z wykorzystaniem języka C i oficjalnego SDK firmy Espressif, czyli ESP-IDF. W tym

momencie posiłkujemy się przykładowym projektem, którym jest nasz ulubiony program Blink. Oczywiście warto też popatrzeć na używaną przez siebie płytkę ESP32 One i jej pinout (rysunek 8.5).



Rysunek 8.5. Pinout płytki ESP32 One²

Ten pinout zapewne nam się przyda. Warto też wiedzieć, gdzie jest jaka dioda, a na tej płytce są aż cztery, umieszczone koło złącza kamerki.

Nas szczególnie interesuje dioda oznaczona *LED1*, która według schematu pokazanego na rysunku 8.6 jest podłączona do *GPIO21*.



ESP32 Pi	FLASH	SRAM	I2S	SPI	DVP	PWM	
S VP BCM26					Y9		GP1036
IO37 BCMI6					Y8		GP1037
IO38 BCMI2					Y7		GP1038
S VN BCM23					Y6		OP1039
IO34 BCM5					Y2		GP1034
IO35 BCM6					Y5		GP1035
IO32 12S0 LRC			I2S WS				GP1032
IO33 12S0 ADC			I2S SDO				
IO25 12S0 DAC					PCLK		
IO26 I2S BCLK			I2S SCK				
IO27 BCM4					HREF		
IO14				SPI CK	Y4		
IO12				SPI SO			
IO13				SPI MO	Y3		
IO15 BCM27				SD CS			
IO2 BCM22							
IO0 BCM25							
IO4 BCM24					XCLK		
IO16		SR CS					
SRAM CLK		SR CK					GPIO17
SHD/SD2	F HO	SR S2					GPIO9
SWP/SD3	F WP	SR S3					GPIO10
SCS/CMD	F CS						GPIO11
SCK/CLK	F CK						GPIO6
SDO/SD0	F DO	SR S0					GPIO7

SDI/SD1	F DI	SR S1					GPIO8
IO5				SPI CS0	VSYNC		
IO18					I2C SDA		
IO23					I2C CLK		
IO19 BCM I7				e CS			
IO22				SPI CS1			
U0RXD TX							GPIO3
U0TXD RX							GPIO1
IO21						LED1	

Rysunek 8.6. Wycinek schematu ESP32 One³

Zwróćcie uwagę na dwie istotne rzeczy. Podczas pracy nad kodem jesteśmy ściśle związani z warstwą sprzętową urządzenia. O ile w przypadku samodzielnych projektów płytek stosujemy własny *schemat*, o tyle gdy używamy gotowego modułu, ważne są dla nas dostarczone przez producenta dokumenty zawierające opis pinów i schemat modułu. Te dwa dokumenty to podstawowe narzędzia podczas pracy z kodem i projektowanym urządzeniem. To jest kolejny, obok nas samych, klucz do labiryntu. Ale przejdźmy do rzeczy...

Już wiemy, że interesująca nas dioda, która w tym przykładzie pokaże nam pierwszą wizualizację naszego kodu w ESP-IDF, jest podłączona do *GPIO21*. Zatem wciskamy w VSC *F1* i wpisujemy:

>ESP-IDF Show Examples Projects

Następnie z listy wybieramy *blink* (rysunek 8.7).

For external components examples, check [IDF Component Registry](#)

get-started

blink

hello_world
sample_project

bluetooth

ble

ble_throughput

throughput_client
throughput_server

ble_ancs

ble_compatibility_test

ble_eddystone

ble_hid_device_demo

ble_ibeacon

Create project using example blink

Blink Example

(See the README.md file in the upper level 'examples' directory for more information about examples.)

This example demonstrates how to blink a LED using GPIO or RMT for the addressable LED, i.e. WS2812.

See the RMT examples in the [RMT Peripheral](#) for more information about how to use it.

How to Use Example

Before project configuration and build, be sure to set the correct chip target using `idf.py set-target`.

Hardware Required

A development board with ESP32/ESP32-S2/ESP32-S3/ESP32-C3 SoC (e.g., ESP32-DevKitC, ESP-WROVER-KIT, etc.)

A USB cable for Power supply and programming

Some development boards use an addressable LED instead of a regular one. These development boards include:

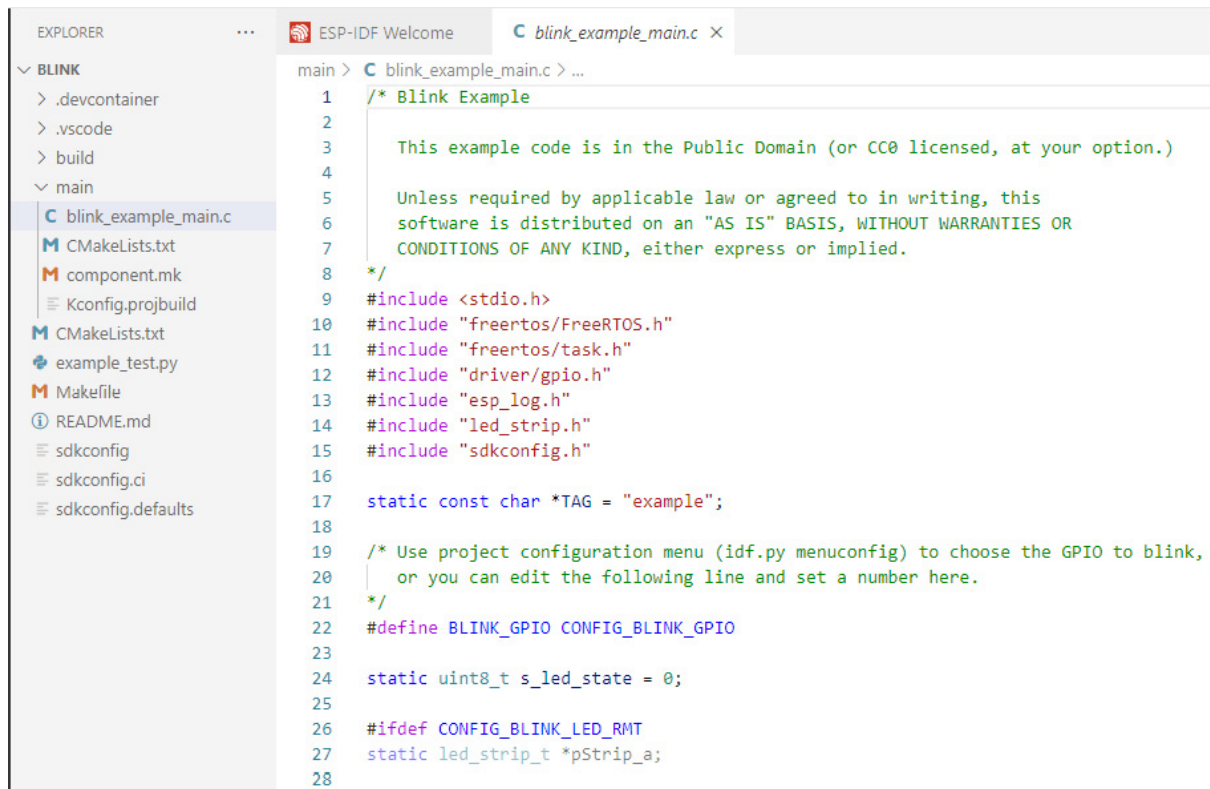
Board	LED type
ESP32-C3-DevKitC-1	Addressable

Rysunek 8.7. Wybór przykładów dla ESP-IDF

Tu dowiadujemy się, do jakich pinów przypisana jest dioda LED w konkretnych zestawach developerskich firmy Espressif, jak program jest skonfigurowany i co pokazuje na wyjściu. Warto się z tym zapoznać, by zrozumieć, jak działa, ale możemy też kliknąć niebieskie pole *Create project using example blink* i wybrać folder dla projektu, który nazwijmy po prostu *blink*.

Tym sposobem został utworzony projekt o nazwie *Blink*. Po lewej stronie okna środowiska mamy dostęp do struktury projektu, jego folderów i plików w drzewie eksploratora projektu. Wybieramy folder *main* i otwieramy nasz plik źródłowy *blink_example_main.c*.

Zawartość pliku wyświetlona w środowisku jest pokazana na rysunku 8.8.



Rysunek 8.8. Podgląd środowiska

Nasz program przedstawia się następująco:

```
/* Blink Example

This example code is in the Public Domain (or CC0
licensed, at your option.)

Unless required by applicable law or agreed to in
writing, this

software is distributed on an "AS IS" BASIS, WITHOUT
WARRANTIES OR

CONDITIONS OF ANY KIND, either express or implied.

*/

#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
```

```

#include "driver/gpio.h"
#include "esp_log.h"
#include "led_strip.h"
#include "sdkconfig.h"
static const char *TAG = "example";
/* Use project configuration menu (idf.py menuconfig)
to choose the GPIO to blink,
or you can edit the following line and set a number
here.
*/
#define BLINK_GPIO CONFIG_BLINK_GPIO
static uint8_t s_led_state = 0;
#ifdef CONFIG_BLINK_LED_RMT
static led_strip_t *pStrip_a;
static void blink_led(void)
{
    /* If the addressable LED is enabled */
    if (s_led_state) {
        /* Set the LED pixel using RGB from 0 (0%) to
255 (100%) for each color */
        pStrip_a->set_pixel(pStrip_a, 0, 16, 16, 16);
        /* Refresh the strip to send data */
        pStrip_a->refresh(pStrip_a, 100);
    } else {
        /* Set all LED off to clear all pixels */
        pStrip_a->clear(pStrip_a, 50);
    }
}
}

```

```

static void configure_led(void)
{
    ESP_LOGI(TAG, "Example configured to blink
addressable LED!");

    /* LED strip initialization with the GPIO and
pixels number*/
    pStrip_a =
led_strip_init(CONFIG_BLINK_LED_RMT_CHANNEL,
BLINK_GPIO, 1);

    /* Set all LED off to clear all pixels */
    pStrip_a->clear(pStrip_a, 50);
}

#elif CONFIG_BLINK_LED_GPIO

static void blink_led(void)
{
    /* Set the GPIO level according to the state (LOW
or HIGH)*/
    gpio_set_level(BLINK_GPIO, s_led_state);
}

static void configure_led(void)
{
    ESP_LOGI(TAG, "Example configured to blink GPIO
LED!");

    gpio_reset_pin(BLINK_GPIO);
    /* Set the GPIO as a push/pull output */
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
}

```

```

#endif

void app_main(void)
{

    /* Configure the peripheral according to the LED
type */
    configure_led();

    while (1) {
        ESP_LOGI(TAG, "Turning the LED %s!",
s_led_state == true ? "ON" : "OFF");
        blink_led();
        /* Toggle the LED state */
        s_led_state = !s_led_state;
        vTaskDelay(CONFIG_BLINK_PERIOD /
portTICK_PERIOD_MS);
    }
}

```

Nie wygląda to dość przyjaźnie, ale w tej chwili nie mamy się czym przejmować. Całość i tak usuwamy, zostawiając tylko linijki `#include`. Całą resztę napiszemy od nowa nieco przejrzyściej. I jak sami zobaczycie — bardzo podobnie do tego, co widzieliście w przykładzie dla Arduino. Nasz program ma być prosty i zrozumiały; oczywiście ważny jest też styl pisanie kodu, w tym wcięcia, które środowisko IDE wprowadza automatycznie, właśnie to bowiem ułatwia późniejsze czytanie i analizę kodu. W takim maluszku nie ma to szczególnego znaczenia, ale gdy kod jest większy —

owszem, i to ogromne. Dlatego popatrzcie, jak powinien wyglądać nasz kod napisany w dobrym stylu:

```
/*
Blink Example
*/
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "esp_log.h"
#include "led_strip.h"
#include "sdkconfig.h"
#define BLINK_GPIO 21
void app_main(void)
{

    gpio_pad_select_gpio(BLINK_GPIO);
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);

    while (1)
    {
        gpio_set_level(BLINK_GPIO, 0);
        vTaskDelay(1000 / portTICK_RATE_MS);
        gpio_set_level(BLINK_GPIO, 1);
        vTaskDelay(1000/ portTICK_RATE_MS);

    }
}
```

Teraz nasz kod wygląda o niebo lepiej, a nawet dwa nieba. Jest prosty i przejrzysty, więc możemy przejść do jego omówienia.

Omówienie programu w ESP-IDF

Omówienie zaczniemy od dołączonych za pomocą instrukcji `#include` plików nagłówkowych. Pisząc w czystym C, musimy dołączyć trochę plików istotnych dla naszego projektu, tak by kompilator rozumiał, co mamy na myśli, gdy używamy funkcji specyficznych dla mikrokontrolerów.

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "esp_log.h"
#include "led_strip.h"
#include "sdkconfig.h"
```

Z tych, które zostawiliśmy z przykładowego programu, kilka tak naprawdę jest niepotrzebnych, więc możemy skrócić kod o kolejne wiersze:

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "sdkconfig.h"
```

Dokładnie tyle nam wystarczy. Pierwsze dwa wiersze informują kompilator, że będziemy korzystać z FreeRTOS, czyli takiego małego systemu operacyjnego, który wiele rzeczy robi za nas. Nie trzeba go używać, ale warto go poznać i się z nim zaprzyjaźnić, gdyż dzięki niemu na pewno łatwiej będzie zapanować nad wieloma aspektami

kodu, zwłaszcza w większych projektach, nie tak prostych jak ten. Kolejna linijka to podstawowy sterownik dla portów GPIO, dzięki któremu możemy ustawiać porty GPIO w szczególności, potrzebny nam w kodzie, sposób, o czym jeszcze powiem. Ostatnia zaś to konfiguracja SDK.

Następną linijką kodu jest:

```
#define BLINK_GPIO 21
```

Określamy w niej nazwę dla GPIO i przypisujemy do niej fizyczny pin GPIO o numerze 21. W zasadzie ta linijka nie jest nam potrzebna, przynajmniej nie w tak małym programie, ale zauważcie, jak jest użyteczna. Wszędzie tam, gdzie będziemy podawać numer pinu, wystarczy użyć zdefiniowanej nazwy BLINK_GPIO. Oczywiście nazwa może być różna, np.:

```
#define LED1 21
```

```
#define przycisk_a 23
```

Jako że nazwy określamy na początku programu, nawet w dużym kodzie w razie potrzeby możemy łatwo i szybko zmienić piny dla różnych urządzeń bez przepisywania numerów w całym kodzie.

Jak pamiętacie, w Arduino ważne były funkcje setup() i loop(). W języku C setup() jako taka nie istnieje, a główną funkcją jest main().

```
void app_main(void)
```

```
{
```

```
    gpio_pad_select_gpio(BLINK_GPIO);
```

```
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
```

```
    while (1)
```

```
    {
```

```

        gpio_set_level(BLINK_GPIO,0);
        vTaskDelay(1000 / portTICK_RATE_MS);
        gpio_set_level(BLINK_GPIO, 1);
        vTaskDelay(1000/ portTICK_RATE_MS);

    }

}

```

W tym przypadku główna funkcja nazywa się `app_main()` i podobnie jak `setup()` w Arduino, jest funkcją typu `void`, czyli nie zwraca żadnych wartości i nie przyjmuje żadnych argumentów. Właśnie ona zawiera pętlę nieskończoną `while()`. Funkcja `main()` w uproszczeniu jest bardzo podobna do `setup()`, gdyż definiujemy w niej ustawienia pinów GPIO i innych rzeczy potrzebnych w programie. Także w jej ciele używamy instrukcji:

```

    gpio_pad_select_gpio(BLINK_GPIO);
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);

```

Pierwsza instrukcja służy do wybrania pinu GPIO i jej argumentem jest wartość `BLINK_GPIO`, która jak wiecie, wynosi 21, czyli jest numerem naszego pinu. Jak więc widzicie, mogliśmy po prostu wpisać tu 21 i program też by działał. Ale jak pisałem, taka forma jest czytelniejsza. Druga instrukcja pozwala nam wybrać kierunek pinu i przyjmuje argumenty w postaci numeru pinu GPIO i trybu pracy, czyli, odpowiednio, `BLINK_GPIO` i `GPIO_MODE_OUTPUT`, dzięki czemu ustaliliśmy, że nasz pin 21 będzie wyjściem. A dokładniej: wyjściem cyfrowym przyjmującym wartości 0 i 1 czyli "low" i "high". I tym sposobem skonfigurowaliśmy piny GPIO. Gdybyśmy teraz resztę kodu umieścili pod konfiguracją pinów zamiast w ciele pętli `while()`, nasz kod wykonałby się tylko raz i zakończył działanie, czyli tak, jak to ma miejsce w `setup()` w Arduino.

```

while (1)
{
    gpio_set_level(BLINK_GPIO,0);
    vTaskDelay(1000 / portTICK_RATE_MS);
    gpio_set_level(BLINK_GPIO, 1);
    vTaskDelay(1000/ portTICK_RATE_MS);
}

```

Pętla główna while() jest podobna do funkcji loop() Arduino. Też jest pętlą nieskończoną i zawarte w niej instrukcje są wykonywane non stop do wyłączenia zasilania układu. W przypadku naszego programu jest to ważne, bo dzięki temu dioda miga cały czas. A dzieje się tak dlatego, że pierwsza instrukcja:

```
gpio_set_level(BLINK_GPIO,0);
```

powoduje ustawienie na pinie 21 stanu niskiego, "low". Następnie funkcja

```
vTaskDelay(1000/ portTICK_RATE_MS);
```

powoduje opóźnienie, czyli oczekiwanie przez czas 1000 ms (1 sekunda) odmierzany przez systemowy timer. Po tym czasie następuje kolejne wywołanie instrukcji:

```
gpio_set_level(BLINK_GPIO, 1);
```

Tym razem instrukcja ustawia dla pinu 21 stan wysoki, 1, czyli "high".

I znów wywołujemy opóźnienie i czekamy 1 sekundę:

```
vTaskDelay(1000/ portTICK_RATE_MS);
```

po czym program wraca na początek i od nowa ustawia stan "low", czeka 1 sekundę, ustawia stan "high", czeka 1 sekundę i tak w kółko, a my obserwujemy migającą diodę LED.

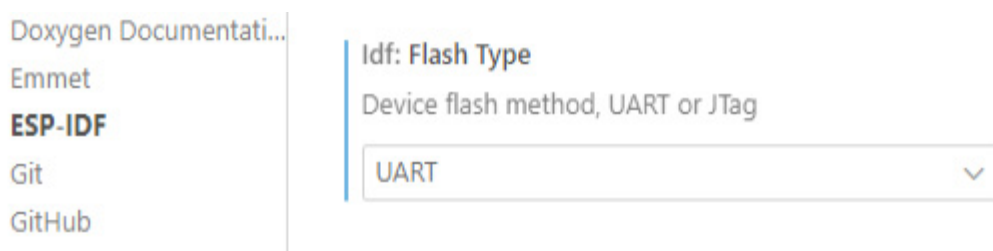
Oczywiście w związku z tym, że pracujemy pod kontrolą RTOS, nasz program jest jednym z wielu tasków, które mają przydzielone określone zasoby i czas wykonania. To wygodne rozwiązanie, dlatego też tu działanie funkcji Delay jest nieco skomplikowane, gdyż nasz task jest zamrażany na 1 sekundę, którą oblicza timer, ale zauważcie, że w międzyczasie może pracować inny task, robiący coś innego... co się przyda w trudniejszych programach. Na przykładzie tego prostego programu doskonale widać wiele podobieństw między Arduino a czystym C; można też zauważyć, jak łatwo się programuje zarówno w Arduino, jak i czystym SDK, nawet pisząc program pracujący pod kontrolą RTOS.

Kompilacja i uruchomienie programu

Powiedzieliśmy już, jak teoretycznie działa nasz program, ale jeszcze nie skończyliśmy. Teraz, zanim zrobimy build, musimy ustawić właściwy port *COM* dla naszej płytki. Wciskamy *F1* i wpisujemy:

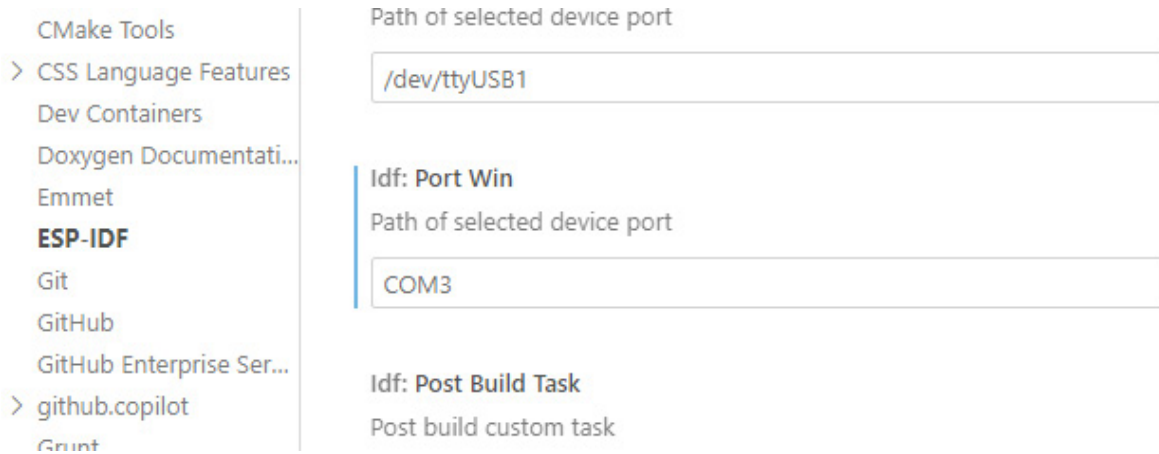
> Preferences Open Settings (UI)

Następnie w nowo otwartym oknie ustawień wybieramy *Extensions/ESP-IDF/* i ustawiamy opcję *Device flash method, UART or JTag*, czyli metodę wgrywania pliku do pamięci flash, na *UART* (rysunek 8.9).



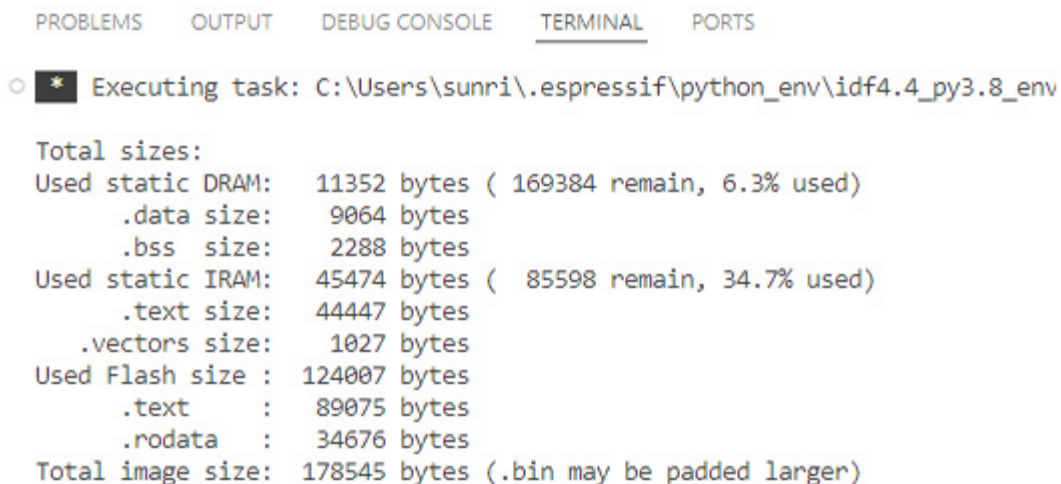
Rysunek 8.9. Ustawienie Flash Type w oknie ustawień

Teraz wybieramy właściwy port *COM* dla naszej płytki. W naszym przypadku płytka zajęła port *COM3* (rysunek 8.10).



Rysunek 8.10. Ustawienie portu COM

Teraz możemy wykonać kompilację naszego programu. Wciskamy *F1* i wpisujemy **ESP-IDF Build your projects**. Po chwili zobaczymy podsumowanie naszego programu (rysunek 8.11).



Rysunek 8.11. Podsumowanie kompilacji

Wszystko przebiegło sprawnie: nasz pierwszy program w ESP-IDF okazał się bardzo prosty pod względem składni języka C i mimo różnic w poleceniach jest bardzo podobny do programu napisanego w Arduino. No, ale zobaczmy, czy działa... Znów wciskamy *F1* i wpisujemy:

```
>ESP-IDF Flash (UART) your projects
```

Po chwili niebieska dioda na naszej płytce radośnie miga co 1 sekundę (rysunek 8.12).





Rysunek 8.12. Widok na płytkę po wgraniu oprogramowania

Podsumowanie

Jak widzicie, wszystko po stronie zarówno Arduino IDE, jak też Visual Studio Code, PlatformIO i oficjalnych narzędzi Espressif przebiegło sprawnie i bez większych problemów. Oczywiście poszliśmy tu na skróty, ale umiecie już skompletować środowisko oraz pracować z różnymi płytkami i ich dokumentacją, dzięki czemu możemy przejść dalej, do — można powiedzieć — właściwego tematu książki, którym jest składanie konkretnych urządzeń. Urządzenia te, zależnie od potrzeb, będziemy oprogramowywać w Arduino lub IDF, od tych najprostszych sterowników IoT, typu ON/OFF, po trudniejsze urządzenia, odczytujące kilka parametrów. Dodatkowo wszystkie nasze urządzenia będą podłączone do internetu, co umożliwi nam sterowanie i odczytywanie parametrów z dowolnego miejsca na ziemi. Dzięki temu stworzymy prawdziwe urządzenia IoT. Proste na początku konstrukcje będziemy obsługiwać przez interfejs Web, a te bardziej skomplikowane będą się komunikować za pomocą protokołu MQTT z brokerem. (Zagadnienie brokera obszerniej omawiam w rozdziale 11.). Tymczasem tych, co chcą zgłębić możliwości układów ESP8266 i ESP32, odsyłam do książek Neila Kolbana (<https://leanpub.com/u/kolban>) o ESP8266 i ESP32. To świetna lektura, dostępna bezpłatnie lub za niewielką opłatą.

2 Źródło: <https://www.waveshare.com/esp32-one.htm>;
dostęp 25.06.2024.

3

Źródło:

https://files.waveshare.com/upload/a/a1/ESP32_One_Sch.pdf;
dostęp 25.06.2024.

CZĘŚĆ III

Wejście do labiryntu

W tym momencie otwarliśmy wrota labiryntu i stoimy w przedsionku. Za wiele nie widać, ale by móc iść nieco dalej, musimy pokrótce wyjaśnić niektóre sprawy i ich znaczenie, gdyż właśnie w odniesieniu do nich będziemy przemierzać korytarze i albo dojdziemy do końca, albo zagubimy się na zawsze. Wiecie już, że jesteście kluczem do labiryntu, poznaliście sprzęt oraz umiecie zainstalować i przygotować środowisko do pracy. Potrafcie też napisać proste programy — zarówno w środowisku Arduino, które ma sporo zalet w postaci rozbudowanych bibliotek i ogromnej społeczności, jak i za pomocą alternatywnych edytorów — a także je zrozumieć. Wreszcie umiecie napisać prosty program z użyciem FreeRTOS w czystym języku C i pod SDK ESP-IDF. Wiecie też, jak wgrać program do pamięci flash, i rozumiecie, jak to wszystko działa. Umiecie pracować ze schematem ideowym i pinologią. To wiele — czas więc pójść na całość. Zanim jednak stworzymy wspólnie pierwsze urządzenia, musicie poznać tajemnice IoT oraz z grubsza zrozumieć protokół MQTT, dzięki czemu będziecie mogli z uniesioną głową przejść przez labirynt...

9. Czym jest IoT

Internet of Things (IoT) to pojęcie, które odnosi się do połączenia urządzeń i przedmiotów z siecią internetową w celu wymiany danych i komunikacji. W ramach IoT urządzenia, takie jak czujniki, maszyny przemysłowe, smartfony czy nawet klasyczne przedmioty codziennego użytku, zyskują zdolność do zbierania, przetwarzania i udostępniania informacji za pośrednictwem internetu. Urządzenia te są często wyposażone we wbudowane sensory, takie jak akcelerometry, czujniki temperatury, wilgotności czy światła, a nawet kamery. Dzięki tym sensorycznym możliwościom mogą zbierać dane z otoczenia lub kontekstu użytkowania i przekazywać je do chmury albo innych urządzeń podłączonych do sieci. Wykorzystując technologie komunikacyjne, takie jak bezprzewodowe sieci, urządzenia IoT mogą w interaktywny sposób komunikować się z innym sprzętem albo oprogramowaniem lub innymi użytkownikami. Na przykład inteligentny system oświetleniowy w budynku może otrzymać informacje o obecności osób i dostosowywać się automatycznie przez włączanie lub wyłączanie światła w odpowiednich miejscach.

Dzięki IoT możliwe staje się zdalne monitorowanie urządzeń i zarządzanie nimi, optymalizacja procesów, automatyzacja zadań oraz tworzenie inteligentnych ekosystemów, które pomagają wydajniej, bezpieczniej i wygodniej funkcjonować zarówno w sferze prywatnej, jak i biznesowej. IoT ma ogromny potencjał do transformacji różnych dziedzin, takich jak przemysł, transport, opieka zdrowotna, energetyka czy rolnictwo. Przyszłość wiąże się z coraz większą liczbą urządzeń podłączonych do sieci oraz

zaawansowaną analizą danych, co pozwala uczynić otoczenie bardziej inteligentnym i zautomatyzowanym. Rozwój technologii IoT wiąże się również z wyzwaniami, takimi jak ochrona prywatności i bezpieczeństwo danych. Zwiększona liczba połączonych urządzeń tworzy większą powierzchnię dla potencjalnych cyberataków, dlatego trzeba je odpowiednio zabezpieczać i chronić pochodzące z nich dane.

Standardy i protokoły komunikacyjne, takie jak MQTT (ang. *Message Queuing Telemetry Transport*) czy HTTP (ang. *Hypertext Transfer Protocol*), umożliwiają komunikację między urządzeniami IoT a platformami chmurowymi lub systemami centralnymi. Wykorzystanie architektur rozproszonych, takich jak architektura oparta na chmurze (ang. *cloud-based*) czy architektura klient-serwer (ang. *client-server*), zapewnia skalowalność i elastyczność w zarządzaniu urządzeniami podłączonymi do sieci. IoT staje się również istotnym elementem w rozwoju koncepcji inteligentnych miast (ang. *smart cities*), gdzie infrastruktura miejska, taka jak systemy transportowe, oświetleniowe, gospodarka odpadami czy monitorowanie jakości powietrza, jest integrowana i zarządzana w sposób efektywny z wykorzystaniem danych generowanych przez różne czujniki wbudowane w urządzenia IoT.

Nasuwa się więc wniosek, że Internet of Things to zaawansowana technologia, która umożliwia urządzeniom łączenie się ze sobą i współpracę, aby mogły generować dane, komunikować się, automatyzować procesy i tworzyć inteligentne środowiska. Technologia ta ma ogromny potencjał do transformacji naszego społeczeństwa i otwiera nowe możliwości w dziedzinie innowacji, efektywności i komfortu życia.

W dalszym rozwoju IoT można zauważyć kilka istotnych trendów. Jednym z nich jest rozwój sztucznej inteligencji

(SI) i uczenia maszynowego, które pozwalają na analizowanie i wykorzystanie ogromnych ilości danych generowanych przez urządzenia IoT. Za pomocą SI możliwe jest rozpoznawanie wzorców, przewidywanie zachowań oraz podejmowanie decyzji na podstawie zebranych informacji. Innym trendem jest rozwój technologii 5G, która zapewnia bardziej zaawansowane i szybsze połączenia bezprzewodowe. To umożliwia przesyłanie większych ilości danych w czasie rzeczywistym, co jest niezbędne do rozwoju bardziej zaawansowanych aplikacji, takich jak samochody autonomiczne czy telemedycyna.

Wraz z rozwojem IoT wzrasta także znaczenie analizy danych (ang. *big data*) i analizy w czasie rzeczywistym (ang. *real-time analytics*). Dane zebrane z urządzeń IoT można analizować w zależności od potrzeb, co prowadzi do lepszych decyzji biznesowych, optymalizacji procesów czy udoskonalenia produktów i usług.

Ważnym aspektem rozwoju technologii IoT jest również zagadnienie zrównoważonego rozwoju. Dzięki wykorzystaniu inteligentnych urządzeń i infrastruktury IoT możliwe jest usprawnienie zarządzania zasobami, redukcja zużycia energii, minimalizowanie ilości odpadów oraz optymalizacja procesów produkcyjnych w celu osiągnięcia większej efektywności ekologicznej.

W przyszłości możemy spodziewać się dalszego wzrostu liczby połączonych urządzeń IoT, wraz z integracją z szerszym ekosystemem technologicznym. IoT będzie odgrywać kluczową rolę w rewolucji przemysłowej, poprawie jakości życia, rozwijaniu inteligentnych miast, a także rozwoju innowacyjnych rozwiązań dla różnych branż i sektorów gospodarki. Powoduje to też wzmożony wzrost nowych zastosowań i scenariuszy wykorzystania tej technologii. Oto kilka przykładów:

- **Zdrowie i opieka zdrowotna:** IoT może przyczynić się do transformacji opieki zdrowotnej przez monitorowanie pacjentów w czasie rzeczywistym za pomocą sensorów, zdalne monitorowanie parametrów życiowych, śledzenie przyjmowania leków oraz zapewnianie spersonalizowanej opieki medycznej. Urządzenia IoT, takie jak inteligentne zegarki czy opaski fitness, mogą pomagać w monitorowaniu aktywności fizycznej, jakości snu i ogólnego stanu zdrowia.
- **Inteligentne domy:** IoT przyczynia się do rozwoju inteligentnych domów, gdzie różne urządzenia domowe są połączone w sieć. Można sterować oświetleniem, ogrzewaniem, klimatyzacją, urządzeniami gospodarstwa domowego oraz systemami bezpieczeństwa, a także je monitorować, za pomocą smartfonów lub głosowych asystentów.
- **Przemysł 4.0:** IoT jest kluczowym elementem rewolucji przemysłowej, w której wyniku urządzenia i maszyny w fabrykach są połączone w sieci, co umożliwia monitorowanie, analizę i optymalizację procesów produkcyjnych. Dzięki temu możliwe jest tworzenie inteligentnych fabryk, gdzie systemy automatyzacji i maszyny współpracują bezpośrednio ze sobą, co skraca czas produkcji i zmniejsza jej koszty.
- **Inteligentne miasta:** IoT ma duże znaczenie w rozwoju inteligentnych miast, gdzie dane z urządzeń IoT są wykorzystywane do zarządzania transportem, parkowaniem, oświetleniem, gospodarką odpadami czy monitorowaniem jakości powietrza. To prowadzi do zwiększenia efektywności, poprawy jakości życia mieszkańców oraz optymalizacji zasobów miejskich.
- **Rolnictwo precyzyjne:** Za pomocą urządzeń IoT można monitorować i automatyzować procesy w rolnictwie, takie jak nawadnianie, nawożenie czy monitoring upraw. Czujniki i urządzenia dostarczają informacji o

wilgotności gleby, jakości powietrza czy pogodzie oraz identyfikują choroby i szkodniki, co umożliwia precyzyjne działania i optymalizację plonów.

Rozwój IoT otwiera również pole do innowacji, tworzenia nowych biznesowych modeli i partnerstw pomiędzy różnymi branżami. Zarazem jednak należy rozważać równowagę między korzyściami a kwestiami bezpieczeństwa, prywatności i etyki związanymi z gromadzeniem i przetwarzaniem danych w środowisku IoT.

Wiecie już, czym jest IoT i jak ogromną rolę odgrywa w naszym świecie. Technologia ta nie jest jednakże zamknięta w laboratoriach i nie jest też domeną wielkich firm technologicznych. Dzięki takim układom jak ESP8266 i ESP32 oraz open-source'owym rozwiązaniom jest dostępna dla każdego. Każdy amator może bezproblemowo tworzyć własne rozwiązania dostosowane do jego potrzeb i dzielić się nimi ze społecznościami, mającymi coraz większy wpływ na postęp techniczny i mnogość urządzeń, które są coraz lepsze, tańsze i bardziej dostępne dla mas. To ogromna odpowiedzialność, prowadząca też do stosunkowo dużych problemów. Ale są one rozwiązywane poniekąd na bieżąco. Teraz już nie dziwią nikogo inteligentne parkingi pozwalające na zautomatyzowane pobieranie płatności w obiektach zamkniętych, oparte na rozpoznawaniu tablic rejestracyjnych, ani wszechobecne tablice informacyjne o czasie przejazdu, możliwych utrudnieniach czy liczbie miejsc parkingowych w miastach. Podobne pole do popisu mamy także my, amatorzy. Możemy stworzyć wiele urządzeń, o których nie pomyśleli inżynierowie, i wdrożyć je we własnych mikrosystemach IoT, od prostych czujników czy włączników oświetlenia, poprzez sterowanie roletami i bramami, po systemy zarządzania klimatem w szklarniach czy hodowlach zwierząt domowych w akwariach i terrariach. Pomyślcie, czy warto — może i Wy macie

genialny pomysł na urządzenie, które nie istnieje lub istnieje, ale jest dostępne w zaporowej cenie. Teraz ta rewolucja dzieje się na naszych oczach i Wy możecie być jej częścią. Możecie stworzyć niewielkim nakładem coś wielkiego, co być może wspomogłoby niepełnosprawnych, oczyści wodę, zminimalizuje zużycie energii we własnym gospodarstwie. Bo tylko od nas zależy, co z tym wszystkim zrobimy. Można by sądzić, że filmowy Skynet jest coraz bliżej, ale on już tu jest i tylko my, amatorzy możemy przyczynić się do jego dynamiczniejszego rozwoju, ale na naszych zasadach, z pominięciem wielkich firm ingerujących w nasze życie prywatne. Myślę, że gra jest warta świeczki. Nie tylko od strony technicznej, bo tworzenie urządzeń przynosi też wiele nauki, a ucząc się przez tworzenie, poznajemy problematykę od podszewki, od pojedynczej diody do sterowania wszystkim, ale również dostrzegamy problematykę bezpieczeństwa i konieczność ochrony własnej prywatności. Zyskujemy świadomość. I możemy mieć wpływ na to wszystko, co nas otacza.

10. Protokół MQTT

MQTT (ang. *Message Queuing Telemetry Transport*) jest lekkim protokołem komunikacyjnym, który opracowano z myślą o efektywnym przesyłaniu danych pomiędzy urządzeniami w architekturze Internetu Rzeczy (IoT). Stworzyli go w 1999 roku Andy Stanford-Clark z IBM-a i Arjun Sage z Eurotechnu. Od tego czasu stał się jednym z najpopularniejszych protokołów w dziedzinie IoT ze względu na swoją prostotę, niskie zużycie zasobów i zdolność do obsługi dziesiątek tysięcy urządzeń jednocześnie. Podstawowym celem MQTT jest umożliwienie komunikacji między urządzeniami przy zminimalizowanym zużyciu energii i przepustowości sieci. Protokół ten zaprojektowano tak, aby przesyłanie danych odbywało się w sposób asynchroniczny i w czasie rzeczywistym, co jest istotne w środowisku IoT, gdzie urządzenia mogą mieć ograniczone zasoby i łączność. MQTT opiera się na architekturze publikuj-subskrybuj (ang. *publish-subscribe*), która umożliwia łatwe i elastyczne wymienianie informacji pomiędzy różnymi źródłami danych a ich odbiorcami. W architekturze tej występują trzy główne elementy:

- **Broker MQTT.** Jest to serwer pośredniczący w wymianie danych pomiędzy urządzeniami. Broker odbiera dane opublikowane przez urządzenia źródłowe i przekazuje je do odpowiednich odbiorców — subskrybentów. Broker jest odpowiedzialny za zarządzanie tematami (ang. *topic*), będącymi nazwami kanałów, na których dane są publikowane i subskrybowane. Broker może być zarządzany w chmurze lub lokalnie, w zależności od konkretnego wdrożenia. W moim przypadku używam rozwiązania

lokalnego, w którym broker MQTT Mosquitto, bezpłatny i opensource'owy, został postawiony na popularnym Raspberry Pi Zero. Za wyborem tej płytki SBC przemawiały nie tylko małe rozmiary, ale też niewielki pobór energii.

- **Wydawca (ang. *publisher*).** Jest to urządzenie, które publikuje dane na temat (*topic*) w brokerze. Wydawca decyduje, które informacje chce udostępnić i w jakim temacie chce je opublikować. Po wydaniu danych broker przekaże je do wszystkich odbiorców subskrybujących ten temat. Przykładem takiego urządzenia wydawcy może być czujnik temperatury i wilgotności powietrza, który zbudujemy w dalszej części.
- **Subskrybent (ang. *subscriber*).** To urządzenie, które subskrybuje (nasłuchuje) określone tematy w brokerze. Subskrybent otrzymuje dane opublikowane w subskrybowanym temacie. Może subskrybować wiele tematów jednocześnie i otrzymywać dane tylko w tych tematach, na które się zarejestrował. Tu przykładem może być panel informacyjny, na którym będzie można obserwować dane z czujników temperatury umieszczonych w różnych miejscach w celu ich monitorowania i/lub użycia do ustawiania odpowiednich parametrów.

Jak więc widzicie, komunikacja w MQTT opiera się na dwóch głównych operacjach: publikowaniu (ang. *publishing*) i subskrybowaniu (ang. *subscribing*). Wydawca publikuje dane w konkretnym temacie, a broker przekazuje te dane do wszystkich subskrybentów subskrybujących ten temat. To jest bardzo wygodne i skuteczne rozwiązanie, przy czym nie wymaga dużych mocy obliczeniowych i ogromnych baz danych koniecznych do przechowywania informacji. Jednak w sytuacji gdy zbierane dane są potrzebne do zestawień statystycznych, np. w celu tworzenia wykresów temperatur i wilgotności oraz

oświetlenia, nic nie stoi na przeszkodzie, by w prosty sposób dane publikowane na brokerze MQTT przekazywać do baz danych i je później prezentować w różnej formie.

Warto jednak wiedzieć, że w MQTT używane są trzy poziomy jakości usługi (ang. *Quality of Service*, QoS), które kontrolują sposób dostarczania danych:

- Poziom **QoS 0** (*At most once*): Oznacza, że dane są dostarczane tylko raz, bez żadnej gwarancji, że docierają do odbiorcy. Jest to najprostszy i najmniej niezawodny sposób dostarczania danych.
- Poziom **QoS 1** (*At least once*): Zapewnia, że dane są dostarczane do odbiorcy co najmniej raz. W razie utraty danych lub braku potwierdzenia dostarczenia broker ponowi próbę wysłania danych.
- Poziom **QoS 2** (*Exactly once*): Zabezpiecza dostarczenie danych dokładnie raz, co wyklucza możliwość ich duplikacji. Jest to najbardziej niezawodny, ale również najbardziej kosztowny pod względem zasobów sposób dostarczania danych.

MQTT może być używany w różnych scenariuszach IoT, takich jak monitorowanie czujników, sterowanie urządzeniami, zdalny dostęp do aplikacji czy gromadzenie danych. Protokół ten wykorzystuje się w wielu aplikacjach, od domowych systemów automatyzacji, poprzez inteligentne miasta, aż po skomplikowane systemy w zastosowaniach przemysłowych.

Podsumowując, MQTT jest protokołem komunikacyjnym wykorzystywanym w Internet of Things, umożliwiającym efektywne przesyłanie danych w czasie rzeczywistym w środowisku publikuj-subskrybuj. O jego popularności w tej dziedzinie decydują prostota, niskie zużycie zasobów i zdolność do obsługi dużej liczby urządzeń.

11. Broker

Z poprzednich rozdziałów dowiedzieliście się, czym jest technologia IoT i jeden z popularniejszych protokołów przesyłania danych, MQTT. Wiecie też, że choć jest prosty, wymaga serwera pośredniczącego, tzw. brokera. W tym rozdziale postaramy się dokonać niekoniecznie oczywistych wyborów. Poznacie brokery pracujące w chmurze i dostępne dla amatorów bezpłatnie lub za niewielką opłatą oraz dowiedziecie się, jak postawić własny.

Każde z tych rozwiązań ma swoje wady i zalety, ale poniekąd wprowadza nas w nowy wymiar automatyki. Proste urządzenia, które zbudujemy na początku, nie będą ich używać — będziemy w nich stosować prostą komunikację przez stronę WWW — ale w miarę omawiania zasad sterowania urządzeniami będziemy stosować coraz bardziej wyrafinowane metody kontrolowania urządzeń i zbierania danych. To wszystko będzie przydatnym usprawnieniem nie tylko naszej codzienności. I choć poznacie tylko niewielką część możliwości, na pewno będziecie mogli własnymi siłami stworzyć coś wielkiego, coś, co ułatwi codzienne życie, a być może usprawni i uratuje świat.

Na dobry początek przyjrzyjmy się dostępnym rozwiązaniom opartym na chmurze, stosunkowo wygodnym, ponieważ dzięki nim nie musimy inwestować w dodatkowe urządzenia i zasoby,

Brokery dostępne online — wady i zalety

Oto wybrane brokery dostępne online:

- HiveMQ (<https://www.hivemq.com/pricing/>) — ten broker jest udostępniony bezpłatnie w ograniczonym stopniu. Bezpłatnie dostępne jest rozwiązanie bezserwerowe (ang. *serverless*), zapewniające podstawowe elementy brokera, idealnie jednak nadaje się do nauki i eksperymentów z MQTT. W podstawowym planie otrzymujemy wspólną platformę MQTT, do 100 klientów nieprzekraczających 10 GB ruchu danych miesięcznie. Dostępne są protokoły MQTT w wersjach 3.1, 3.1.1 i 5.0, bezpieczne połączenie przez TLS/SSL, dostajemy też podstawowe zasady autoryzacji i obsługę gniazd internetowych. Może nie jest to wiele, ale na początek do zabawy może wystarczyć. Rozwiązanie to ma też swoje wady, np. nie ma gwarancji dostępności usług, a obsługa techniczna jest na bardzo podstawowym poziomie.
- EMQX (<https://www.emqx.com/en/mqtt/public-mqtt5-broker>) — to globalny klaster o dostępie publicznym, a co za tym idzie, każdy może publikować, co chce, i może też widzieć publikacje innych użytkowników.
- MyQttHub (<https://myqttHub.com/en/>) — tu również w bezpłatnym planie otrzymujemy całkiem spore możliwości: do 100 urządzeń, 3 panele, maks. 50 równoczesnych połączeń, 250 subskrypcji, maksymalny rozmiar pojedynczej wiadomości 4 kB. Dość restrykcyjne jest natomiast podejście do wysyłania wiadomości: 300 na minutę, 800 na godzinę, 10 000 na dzień, 310 000 na miesiąc. Ponadto plan zapewnia SSL/TLS, API HTTPS i dostęp do forum.

To tylko wybrane, najbardziej popularne rozwiązania. Jest ich więcej i każdy znajdzie coś dla siebie, zarówno wśród bezpłatnych, jak i płatnych. Myślę, że na początek dobrym wyborem do zabawy i testów będzie pierwsza pozycja na

liście, czyli HiveMQ. Jest dość przejrzysty i oferuje chyba w miarę dobre warunki na początek drogi.

Oczywiście nie można nie wspomnieć o darmowej polskiej Supli. Oferuje darmowy broker, aplikację na Androida i możliwość budowy własnych urządzeń opartych na opensource'owym API, które można zintegrować z Arduino. Supla jest dostępna do pobrania na stronie <https://www.supla.org/pl/>. Umożliwia budowanie własnych urządzeń opartych na udostępnionym kodzie, ale też zakup gotowych urządzeń dostępnych w ofercie firmy Zamel. A od niedawna może być podłączona do dowolnego brokera MQTT. Supla ma tę przewagę, że udostępnia wygodne API do sterowania zgodnymi urządzeniami przez wygodną aplikację i panel sterowania na komputerze. Nowe wydanie clouda Supli zapewnia też integrację z Amazon Alexa i Google Home, ponadto są dostępne sceny pozwalające na ustawienie scenariuszy dla oświetlenia i automatyczne zamykanie bram po określonym czasie od otwarcia. Wiele się zmieniło, od kiedy sam używałem Supli, niemal na początku jej istnienia, w czasach gdy w Zamelu dział Supli dopiero powstawał. Na początku była dla mnie bardzo fajna i wygodna — nie musiałem się bawić we własne aplikacje na telefon i tym podobne, a urządzenia były stosunkowo proste, bo oferowały tylko kilka włączników, czujnik temperatury i sterowanie roletami. Biblioteka dla własnych urządzeń pod Arduino była fajnie napisana, choć stosunkowo opasła i czasami trudno było się dokopać do funkcji. To oczywiście nie zbrodnia, bo Supla jest przykładem naprawdę bardzo dobrego podejścia do projektu skutecznego i wygodnego systemu *smart home*. Wprawdzie urządzenia firmy Zamel mogłyby być tańsze, ale są, i to na dodatek nasze, polskie, z prostą konfiguracją zapewniającą natychmiastowe rezultaty. Do tego mają ładny interfejs, zarówno w panelu WWW, jak i w aplikacji dla Androida (rysunek 11.1).

17:01

16%



supla

Oświetlenie krewetek



Włącznik światła



Włącznik światła

Rolety 1



Roleta

SunRGBW-AQUA



SunWS2812



Kanały



Grupy



Sceny



Rysunek 11.1. Wygląd interfejsu Supla

Ja jednak chciałem więcej. Pisanie niektórych rzeczy pod Suplę nie było łatwe, dlatego postawiłem na rozwiązanie lokalne, chcąc uniezależnić sterowanie domem od działania internetu. Bo niestety, moi drodzy, wszystkie opisane wyżej rozwiązania mają podstawową wadę: nie działa internet, nie mamy kontroli. To spowodowało, że zacząłem szukać lepszej opcji, która byłaby skrojona pod moje zasady i moje rozwiązanie.

Mosquitto — opensource'owy bezpłatny broker

Broker MQTT Mosquitto jest oprogramowaniem open source, co oznacza, że jest dostępny na zasadach otwartych licencji. Mosquitto jest dystrybuowany na podstawie licencji Eclipse Public License (EPL) w wersji 1.0. Licencja ta jest otwartą licencją oprogramowania, zapewniającą użytkownikom pewne prawa, takie jak możliwość modyfikowania, kopiowania i redystrybucji oprogramowania. Jednakże użytkownicy są zobowiązani do przestrzegania warunków określonych w tej licencji. Jako jedna z implementacji protokołu MQTT, został napisany w języku C, dostępnym dla wielu platform, jak Linux, Windows, macOS, oraz różnych architektur, od x86 po ARM. Mosquitto ma też inne zalety, m.in. lekkość i wydajność działania. To wszystko umożliwia dostosowanie go do różnych środowisk i urządzeń, nawet o mocno ograniczonych zasobach sprzętowych. Nie tylko można łatwo go zainstalować i uruchomić na wielu systemach operacyjnych, jest też kompatybilny z wieloma klientami MQTT, dzięki czemu możemy korzystać z różnych rozwiązań. Udostępnia ponadto mechanizmy

bezpieczeństwa SSL/TLS, jak też szereg funkcji i usług, które naprawdę mogą się przydać:

- *Last Will and Testament* — jakkolwiek śmiesznie to brzmi, funkcja umożliwia ustawienie "ostatniej woli", czyli wiadomości, która zostanie automatycznie opublikowana, gdy klient utraci połączenie. Może być przydatna do monitorowania dostępności urządzeń.
- Obsługa *Retained Messages* — to również istotna funkcja, która powoduje zatrzymanie wiadomości i tym samym zapewnia, że nowi subskrybenci ją otrzymają, gdy tylko się zarejestrują w systemie.
- *Bridge Connection* — umożliwia łączenie różnych brokerów MQTT ze sobą.
- *WebSockets* — obsługa protokołu WebSocket. Umożliwia łatwe integrowanie MQTT z aplikacjami opartymi na przeglądarkach.
- Monitorowanie i dzienniki zdarzeń — znacząco ułatwiają śledzenie aktywności i diagnozowanie problemów.
- Społeczność — dzięki otwartości projektu aktywna społeczność developerów dynamicznie rozwija projekt oraz udziela wsparcia użytkownikom.

To takie mniej oczywiste zalety, o których może nie wiedzieliście. Rozwiązanie jednak ma też swoje wady. Może nie są zbyt problematyczne, niemniej warto zwrócić na nie uwagę:

- Nie ma wbudowanych mechanizmów replikacji danych, co może utrudnić np. zwiększenie dostępności systemu przez redundancję.
- W odróżnieniu od komercyjnych rozwiązań, zawiera ograniczoną liczbę zaawansowanych funkcji i narzędzi do zarządzania brokerem i monitorowania go.
- Interfejs nie jest zbyt rozbudowany w porównaniu z rozwiązaniami komercyjnymi, więc w niektórych

przypadkach zarządzanie złożonymi i dużymi sieciami z różnymi poziomami komunikacji może być wyzwaniem.

Jest też sporo ograniczeń licencyjnych, ale o tym przeczytacie w licencji dostępnej z pakietami instalatora.

Tak czy siak, Mosquitto jest bardzo użytecznym i bardzo dobrze wykonanym brokerem, który na pewno jest lepszym rozwiązaniem od pracujących w sieci darmowych publicznych brokerów. Zapewnia nie tylko efektywne narzędzia do implementowania komunikacji opartej na protokole MQTT, ale też wartościowe doświadczenia edukacyjne dla każdego, kto chce poznać IoT oraz komunikację M2M (ang. *machine to machine*), czyli komunikację między maszynami, pozwalającą na transmisję zarówno przewodową, jak i bezprzewodową czy programowanie sieciowe.

Narada MQTT — broker dla Androida

To w zasadzie taka ciekawostka — broker dla Androida o prostym interfejsie, ale w miarę działający, który możemy uruchomić na starszym telefonie czy też tablecie z Androidem, by dać mu drugie życie i tym samym zredukować ilość elektrośmieci. Takie podejście nazwałem sobie ekoelektroniką, czyli elektroniczną ekologią. Oczywiście tak spreparowane urządzenie może też pełnić funkcję WallPanela, obrazując i kontrolując dane MQTT poprzez łatwo dostępne oprogramowanie klienckie typu dashboard. Naradę znajdziecie na GitHubie pod adresem <https://github.com/eswarm/narada>, możecie też zainstalować gotowca ze sklepu Google Play, dostępnego na stronie <https://play.google.com/store/apps/details?id=in.eswarm.narada>.

Oprogramowanie działa i jest bardzo proste w konfiguracji, co może być atutem na początku naszej zabawy. Jest opensource'owy, korzysta z brokera Moquette i obsługuje MQTT 3.1. Można go więc potraktować jako narzędzie edukacyjne, broker do małych zadań i niewielkich sieci oraz jako ciekawostkę.

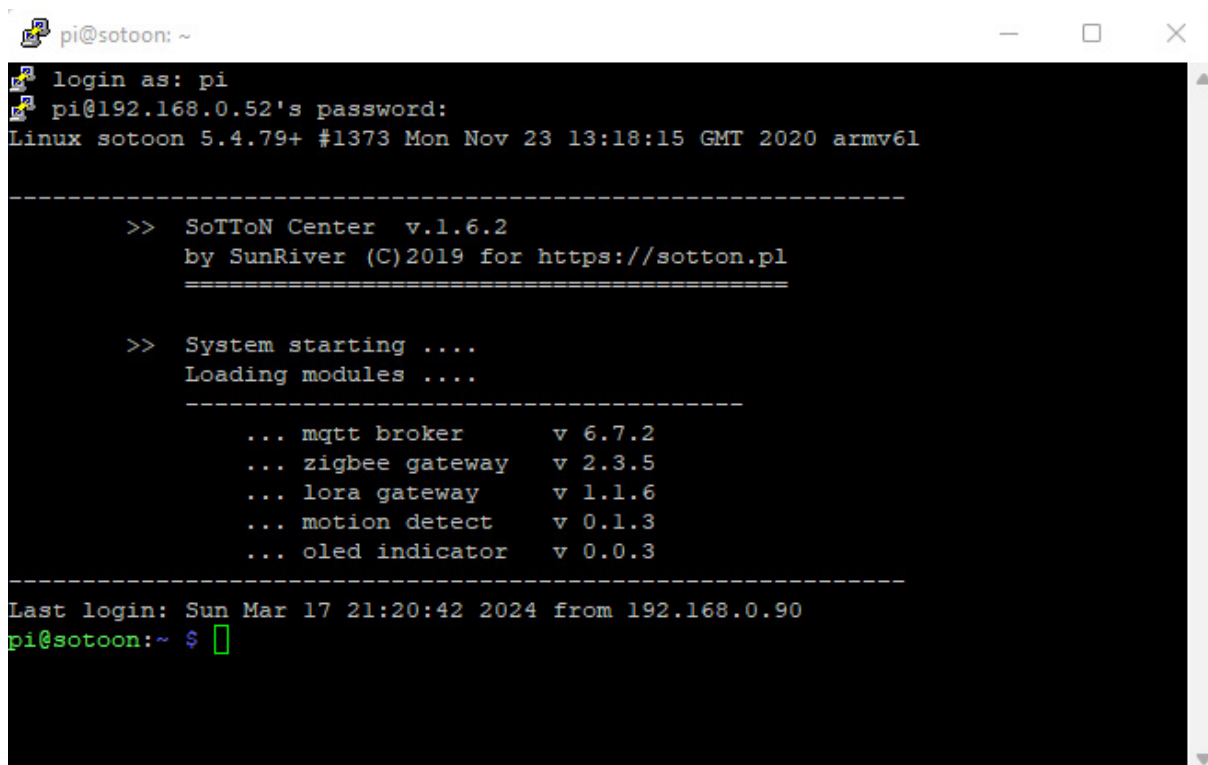
12. Stawiamy własny lokalny broker MQTT

Którąkolwiek opcję z przedstawionych w poprzednim rozdziale wybierzemy, warto wiedzieć, jak postawić własny, lokalny broker, który pozwoli kontrolować swój system niezależnie od tego, czy działa internet, czy też od dostępności bezpłatnych usług na publicznych brokerach. W opisie oprę się na łatwo dostępnym rozwiązaniu SBC (ang. *Single Board Computer*). I tym razem nie jest istotne samo rozwiązanie, którym może być popularne Raspberry Pi, Khadas, Odroid czy nasze polskie SomLabs, Orange Pi, a nawet tani ThinTerminal. Ważne jest, by nasze urządzenie, które będzie pełnić ową zaszczyną funkcję, charakteryzowało się niewielkim rozmiarem i względnie małym zapotrzebowaniem na energię oraz zapewniało połączenie z siecią, czy to przez WiFi, czy też za pomocą kabla Ethernet. Idealne do tego celu może być właśnie rzeczony Raspberry Pi lub którykolwiek alternatywny mikrokomputer. Sam zdecydowałem się na Mosquitto, który zainstaluję na jednym ze swoich urządzeń SBC.

Testowo wybór padł na Khadas VIM2 Pro. Wiem, szaleństwo, osiem rdzeni... Akurat on wydawał mi się najmniej użyteczny do innych zadań w kontekście pozostałych SBC i rozwiązań w stylu Raspberry Pi, ale do roli brokera — jak najbardziej. Dodatkowym atutem Khadasa jest dla mnie to, że ma pamięć EMMC, nośnik nieco pewniejszy od kart SD, z których trwałością różnie bywa podczas ciągłej pracy, podczas gdy zarówno PineA64, jak i SomLABS używają, podobnie jak Raspberry Pi, systemu zainstalowanego właśnie na karcie SD. Oczywiście

każdy powinien wybrać sprzęt, który woli lub posiada, ja wybrałem Khadasa. Opis instalacji przebiega tak samo we wszystkich rodzajach SBC z systemem Linux.

W swoim SBC postawiłem system Linux Ubuntu MATE, ale jak wspomniałem, może to być każdy inny Linux, np. Raspbian. Mosquitto, jak już wiecie, wybrałem nie bez powodu; będzie doskonale się spisywać na małym SBC, który notabene większość czasu nie będzie wykonywał żadnych zadań, ale co tam. Sama instalacja brokera nie jest jakoś specjalnie trudna i nie nastręczy wielu problemów. Wystarczy zalogować się na terminal, czy to lokalnie na SBC, czy też przez SSH, jak to ja robię, bo nie chce mi się podpinąć do khadasa monitora i klawiatury. Następnie w terminalu wykonujemy kilka poleceń (rysunek 12.1).



```
pi@sotoon: ~  
login as: pi  
pi@192.168.0.52's password:  
Linux sotoon 5.4.79+ #1373 Mon Nov 23 13:18:15 GMT 2020 armv6l  
  
-----  
>> SoTTon Center v.1.6.2  
by SunRiver (C)2019 for https://sotton.pl  
=====
```

>> System starting
Loading modules

... mqtt broker v 6.7.2
... zigbee gateway v 2.3.5
... lora gateway v 1.1.6
... motion detect v 0.1.3
... oled indicator v 0.0.3

```
Last login: Sun Mar 17 21:20:42 2024 from 192.168.0.90  
pi@sotoon:~ $
```

Rysunek 12.1. Widok połączenia SSH z systemem Linux

Podstawą jest aktualizacja pakietów, więc wpisujemy:

```
$> sudo apt-get update
```

Czekamy chwilę, i możemy przystąpić do instalacji brokera, co robimy poleceniem:

```
$> sudo apt-get install mosquitto
```

Tu, zależnie od szybkości łącza, zajmie to dłuższą chwilę lub dwie. Po drodze instalator zapyta o doinstalowanie koniecznych bibliotek, na co wyrażamy zgodę wciśnięciem klawisza Y. I tym sposobem nasz broker zostanie wkrótce zainstalowany. Teraz do testów przyda nam się jakiś klient, więc wpisujemy polecenie:

```
$> sudo apt-get install mosquitto-clients
```

I po chwili wszystko jest gotowe. Teraz zatem wypada nam sprawdzić, czy nasz broker działa poprawnie. Musimy się z nim jakoś połączyć, dlatego uruchamiamy drugą instancję terminala, np. przez kliknięcie *term* na ekranie podłączonym do SBC, jeśli pracujemy na nim bezpośrednio z użyciem monitora i klawiatury, albo poprzez SSH, przez uruchomienie kolejnego okienka terminala Putty.

U mnie wygląda to tak jak na rysunku 12.2. Jak widzicie, są uruchomione dwa okna terminala i tu chodzi o to, że w jednym terminalu będziemy publikować wiadomości testowe, a w drugim je subskrybować.

The image displays two terminal windows, one above the other, both showing the same sequence of commands and output. The terminal is titled 'pi@sotoon: ~'. The user logs in as 'pi' with the IP address '192.168.0.52'. The system is Linux sotoon 5.4.79+ #1373 Mon Nov 23 13:18:15 GMT 2020 armv61. The user enters the command to start 'SoTtoN Center v.1.6.2' by SunRiver (C)2019 for https://sotton.pl. The system starts, loading modules for mqtt broker (v 6.7.2), zigbee gateway (v 2.3.5), lora gateway (v 1.1.6), motion detect (v 0.1.3), and oled indicator (v 0.0.3). The last login is recorded as Sun Mar 17 17:28:44 2024 from 192.168.0.90. The prompt is 'pi@sotoon:~ \$'.

```
pi@sotoon: ~
login as: pi
pi@192.168.0.52's password:
Linux sotoon 5.4.79+ #1373 Mon Nov 23 13:18:15 GMT 2020 armv61

>> SoTtoN Center v.1.6.2
    by SunRiver (C)2019 for https://sotton.pl
=====

>> System starting ....
    Loading modules ....

    ... mqtt broker      v 6.7.2
    ... zigbee gateway   v 2.3.5
    ... lora gateway     v 1.1.6
    ... motion detect    v 0.1.3
    ... oled indicator   v 0.0.3

Last login: Sun Mar 17 17:28:44 2024 from 192.168.0.90
pi@sotoon:~ $
```

The second terminal window shows the same output, but the last login time is updated to Sun Mar 17 21:20:19 2024 from 192.168.0.90.

```
pi@sotoon: ~
login as: pi
pi@192.168.0.52's password:
Linux sotoon 5.4.79+ #1373 Mon Nov 23 13:18:15 GMT 2020 armv61

>> SoTtoN Center v.1.6.2
    by SunRiver (C)2019 for https://sotton.pl
=====

>> System starting ....
    Loading modules ....

    ... mqtt broker      v 6.7.2
    ... zigbee gateway   v 2.3.5
    ... lora gateway     v 1.1.6
    ... motion detect    v 0.1.3
    ... oled indicator   v 0.0.3

Last login: Sun Mar 17 21:20:19 2024 from 192.168.0.90
pi@sotoon:~ $
```

Rysunek 12.2. Otwarte dwa okna terminalowe

Zatem w pierwszym oknie wpisujemy:

```
$> mosquitto_sub -t "testmqtt"
```

To polecenie uruchomi klienta subskrypcji. Wskazujemy w nim parametrem `-t`, że zapisaliśmy się do subskrybowania wiadomości, które nadejdą w wybranym przez nas temacie, czyli *testmqtt*. Teraz gdy nasz klient oczekuje wiadomości, w drugim oknie terminala wpisujemy:

```
$> mosquitto_pub -m "Witam na naszym małym serwerku  
MQTT" -t "testmqtt"
```

To polecenie spowoduje wysłanie wiadomości Witam na naszym małym serwerku MQTT do tematu *testmqtt*. Jak zauważyliście, nasza wiadomość dotarła do pierwszego okienka, gdyż została odebrana przez uruchomionego tam klienta. Tak więc wszystko działa wyśmienicie. Możemy np. łatwo dołączyć do całości prosty termometr oparty na ESP8266 (który później zbudujemy) i będzie on publikował temperaturę otoczenia (rysunek 12.3).

A terminal window titled 'pi@sotoon: ~' with standard window controls. The terminal displays a list of temperature readings in Celsius, such as 6.80, 6.90, and 6.70, arranged in a single column. A green cursor is visible at the bottom left of the terminal area.

```
pi@sotoon: ~
6.80
6.80
6.90
6.80
6.80
6.80
6.80
6.80
6.90
6.90
6.80
6.80
6.70
6.70
6.70
6.70
6.70
6.70
6.70
6.80
6.80
6.80
```

Rysunek 12.3. Odczyt temperatury w terminalu

Oczywiście to mało eleganckie rozwiązanie, ale obrazuje, że wszystko działa poprawnie. Dane zaś możemy sobie obrazować w którymkolwiek z łatwo dostępnych paneli, np. MQTTDash, dostępnym w sklepie Google Play. Oczywiście w tej chwili jest to bardzo prosta komunikacja, otwarta, bez jakichkolwiek zabezpieczeń. Dlatego wypadaloby nieco skonfigurować nasz broker, skoro mamy pewność, że wszystko działa jak należy. Podstawowym zabezpieczeniem będzie hasło wymagane do zalogowania się do brokera. Rzecz jasna w pakiecie Mosquitto mamy dostępne odpowiednie narzędzie. Wystarczy wpisać:

```
$> sudo mosquitto_passwd -c /etc/mosquitto/passwd  
<nazwa brokera>
```

Teraz po wyświetleniu się zachęty Password: wpisujemy hasło i potwierdzamy klawiszem *Enter*. Łatwo i przyjemnie,

jednak to rozwiązanie ma wadę. Nasze hasło jest pamiętane tylko do ponownego uruchomienia systemu lub samego brokera, ale jest na to stosunkowo prosta rada — tym samym nauczycie się podstaw pracy w terminalu bash Linuxa. Zatem musimy wprowadzić za pomocą jakiegoś edytora zmiany w pliku konfiguracyjnym. W terminalu najwygodniej użyć nano, co robimy poleceniem:

```
$> sudo nano /etc/mosquitto/conf.d/default.conf
```

Utworzy ono pusty dokument tekstowy, w którym dodamy dwa wpisy:

```
allow_anonymous false  
password_file /etc/mosquitto/passwd
```

Pierwszy wiersz uniemożliwia połączenia anonimowe do brokera, a drugi informuje broker o miejscu przechowywania hasła.

Teraz musimy zapisać plik i zamknąć edytor nano. (Swoją drogą — prawda, że przyjemny?) Żeby zapisać plik, wciskamy *Ctrl+O* i potwierdzamy zapis klawiszem *Enter*, a następnie zamykamy nano kombinacją *Ctrl+X*. Jeśli nie mieliście dotąd styczności z konsolą Linuxa, to sami widziecie na tym prostym przykładzie, że wcale nie jest taka straszna i daje się przyjemnie pracować. Teraz musimy zrestartować broker — oczywiście można to szybko zrobić przez odłączenie zasilania SBC, ale zrobmy to porządnie, jak na programistów przystało. Wpisujemy więc w terminalu:

```
$> sudo systemctl restart mosquitto
```

I już... Teraz musimy się tylko przelogować w oknie subskrybenta. I tak bowiem będzie niepołączony, gdyż wymagane jest wpisanie hasła. Żeby to zrobić, wciskamy *Ctrl+C* i ponownie uruchamiamy subskrybenta, ale już nieco inaczej:

```
$> mosquitto_sub -t "testmqtt" -u "<nasza nazwa>" -P "  
<hasło>"
```

Co ważne, hasło podajemy po parametrze -P (zwróćcie uwagę na wielkość litery). Teraz w dolnym oknie możemy opublikować jakąś wiadomość... Ojoj... system ją odrzucił?

```
Connection Refused: not authorised.
```

```
Error: The connection was refused.
```

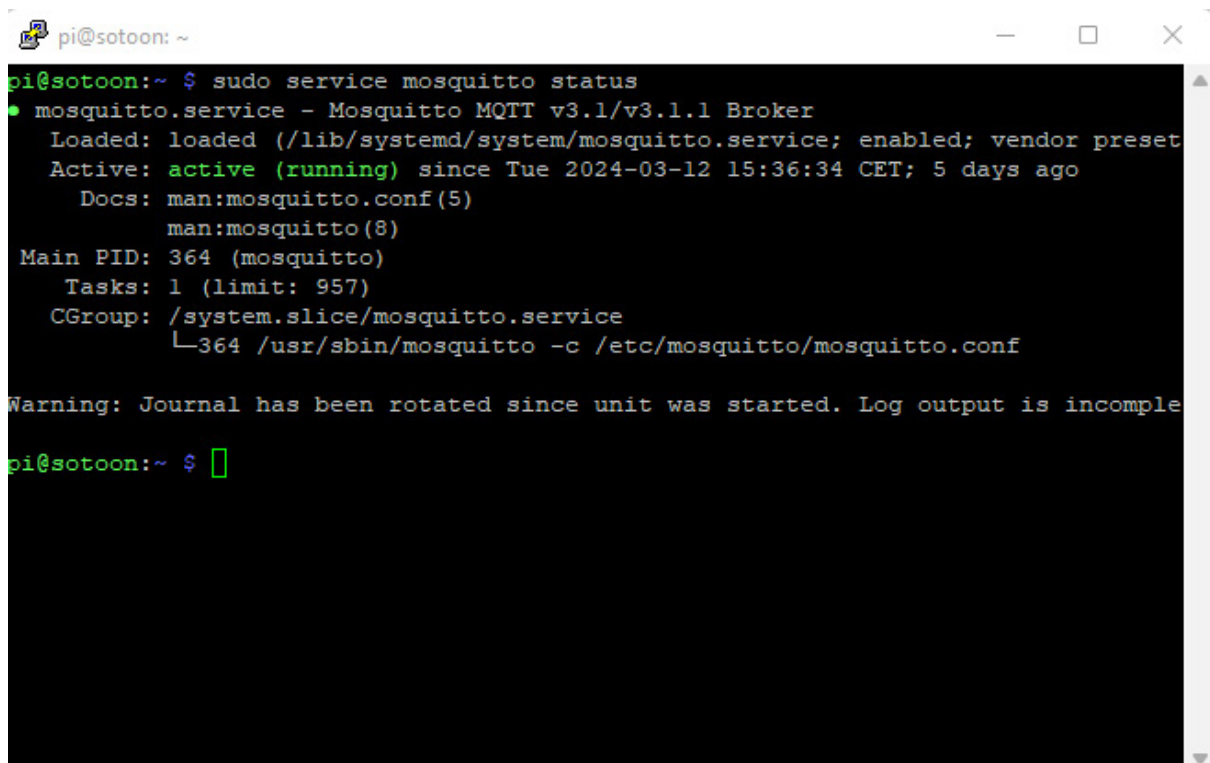
Właśnie tak, nasz wydawca nie został autoryzowany, a tylko autoryzowani wydawcy i subskrybenci mogą od tej pory wysyłać i odbierać wiadomości. O ile autoryzowaliśmy subskrybenta, to teraz musimy autoryzować wydawcę, co robimy tak:

```
$> mosquitto_pub -t "testmqtt" -m "Poprawnie się  
zalogowałem do tematu" -u"<nasza nazwa>" -P "<hasło>"
```

Widzicie? Teraz działa i tak od teraz będzie się odbywało logowanie poprzez ustalone hasło, zarówno w przypadku publikacji, jak i subskrypcji. Łączenie się przez SSL/TLS w tej chwili nie jest nam potrzebne, ponieważ broker działa lokalnie w naszej sieci, więc teraz możemy przystąpić do eksperymentów. Istotne w tym momencie jest sprawdzenie statusów brokera, co wykonujemy poleceniem:

```
$> sudo service mosquitto status
```

Rezultat wykonania tego polecenia jest pokazany na rysunku 12.4.

A terminal window titled 'pi@sotoon: ~' with standard window controls. It displays the output of the command 'sudo service mosquitto status'. The output shows that the 'mosquitto.service' is loaded, enabled, and active (running) since Tuesday, 2024-03-12 at 15:36:34 CET, 5 days ago. It lists the main PID as 364, tasks as 1, and the CGroup as /system.slice/mosquitto.service. A warning message at the bottom states: 'Warning: Journal has been rotated since unit was started. Log output is incomplete'.

```
pi@sotoon:~ $ sudo service mosquitto status
● mosquitto.service - Mosquitto MQTT v3.1/v3.1.1 Broker
   Loaded: loaded (/lib/systemd/system/mosquitto.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2024-03-12 15:36:34 CET; 5 days ago
     Docs: man:mosquitto.conf(5)
           man:mosquitto(8)
  Main PID: 364 (mosquitto)
    Tasks: 1 (limit: 957)
   CGroup: /system.slice/mosquitto.service
           └─364 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

Warning: Journal has been rotated since unit was started. Log output is incomplete.

pi@sotoon:~ $
```

Rysunek 12.4. Status brokera Mosquitto

I to by było na tyle. Broker działa i możemy do niego dołączać dowolne urządzenia IoT poprzez sieć. Oczywiście to tylko podstawy, a jako że zajmujemy się IoT, skupiliśmy się wyłącznie na podstawach łączenia się przez sieci WiFi i internet. Ale możliwości mamy znacznie więcej...

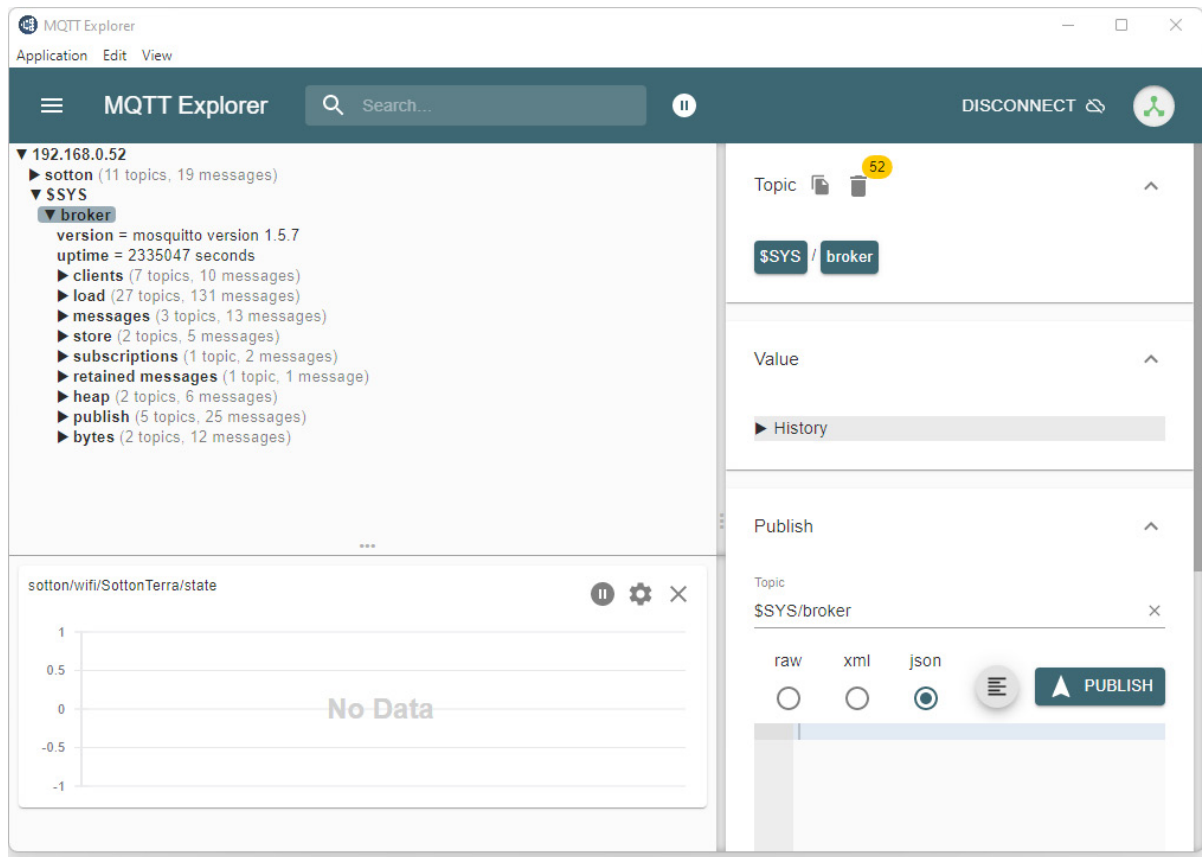
Przydatne narzędzia do monitorowania brokera

Gdy już będziemy tworzyć urządzenia, które będziemy podłączać do swojego brokera, nieodzowne może się okazać oprogramowanie narzędziowe, dzięki któremu szybko i łatwo przejrzymy, co się dzieje na naszym brokerze i czy dane są publikowane tak, jak chcieliśmy, w tematach, w których chcieliśmy je publikować. Będzie też można

sprawdzić, czy opublikowanie danej wiadomości w temacie urządzenia wywołuje reakcję. Oczywiście możemy to obserwować w terminalu, ale jest to mało wygodne; czasem wymaga użycia nawet kilku okien terminala. Z początku chciałem sam napisać takie narzędzie, ale znalazłem bardzo dobry i wygodny program MQTT Explorer autorstwa Thomasa Nordquista. Znajdziesz go na stronie projektu pod adresem <http://mqtt-explorer.com/>. Aplikacja ta działa pod wszystkimi systemami, jest prosta i bardzo wygodna. Ja używam wersji przenośnej tego narzędzia.

Udostępnia ono podgląd ruchu na brokerze oraz wszystkich danych publikowanych w tematach, a także pozwala przeglądać wykresy i łatwo publikować w dowolnym temacie, co z kolei umożliwia łatwe zweryfikowanie poprawności działania swoich urządzeń oraz sekwencji danych.

Przydatne dane systemowe



Rysunek 12.5. MQTT Explorer — widok okna programu

Nasz broker poza oczywistymi informacjami publikuje szereg danych systemowych, które mogą być nam przydatne na wiele sposobów (rysunek 12.5). Oczywiście, jeśli wcześniej Was zaciekałem i zgłębiliście dokumentację brokera, nie dowiecie się tu nic odkrywczego. Ale jeśli ktoś był leniuszkiem, to spokojnie dowie się tego i owego, co być może mu się przyda. Gdy podglądaliście broker MQTT Explorer, pewnie zauważyliście temat systemowy `#SYS`. W tym temacie wszystkie dane są umieszczane z parametrem `-r`, co powoduje, że każdy subskrybent otrzymuje ostatnią aktualną informację. Z tych bardziej nam przydatnych danych można wymienić:

- wersję brokera,
- timestamp,
- uptime.

Pozyskać te informacje możemy podobnie jak wszystkie inne. Wystarczy zasubskrybować właściwy temat przez wpisanie w terminalu np.:

```
$>mosquitto_sub -t '$SYS/broker/version' -u <nazwa> -P  
<hasło>
```

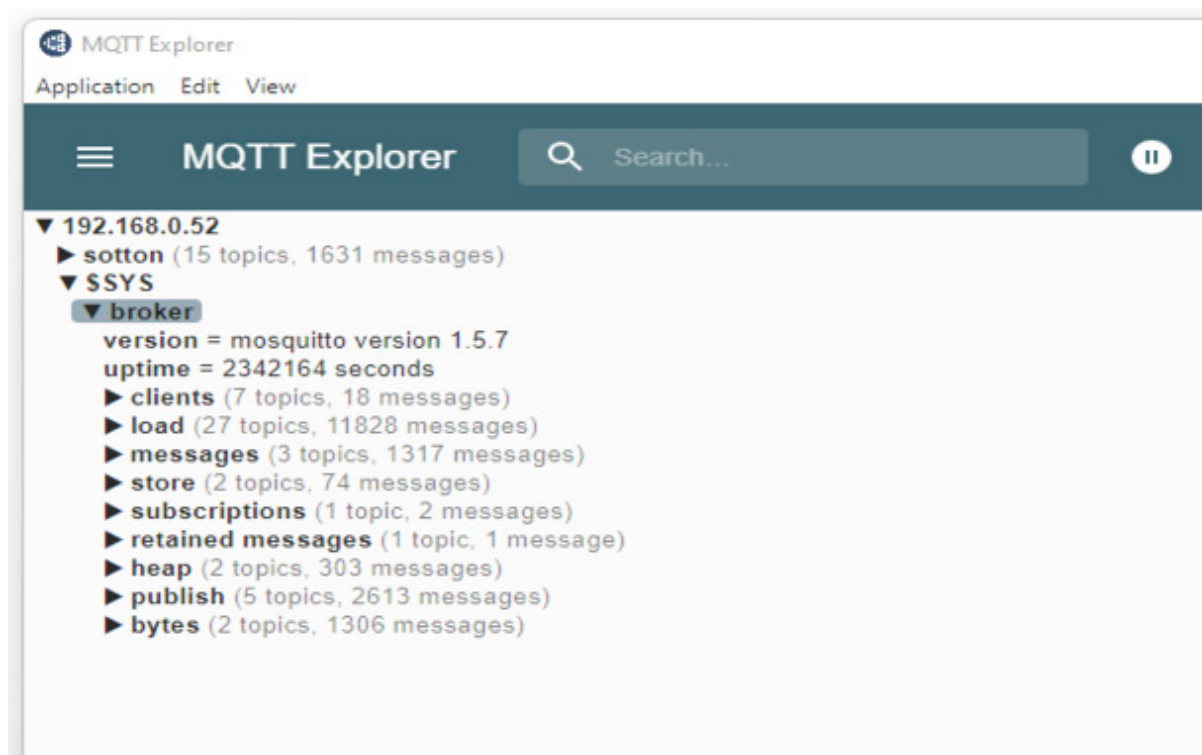
I w ten sposób uzyskaliśmy wersję brokera. W podobny sposób możemy wydostać informację o czasie działania naszego serwera bez restartu:

```
$>mosquitto_sub -t '$SYS/broker/uptime' -u <nazwa> -P  
<hasło>
```

To niezwykle przydatna informacja, która pozwala tworzyć statystyki dotyczące awarii brokera lub restartów systemu spowodowanych chwilowym brakiem zasilania podczas naszej nieobecności. Tu jednak Was rozczaruję, jeśli przez chwilę pomyśleliście, że skoro dane są w czasie rzeczywistym, to można będzie ich użyć jako timera sekundowego. Nie, nie można, z tej prostej przyczyny, że uptime jest aktualizowany co 11 sekund.

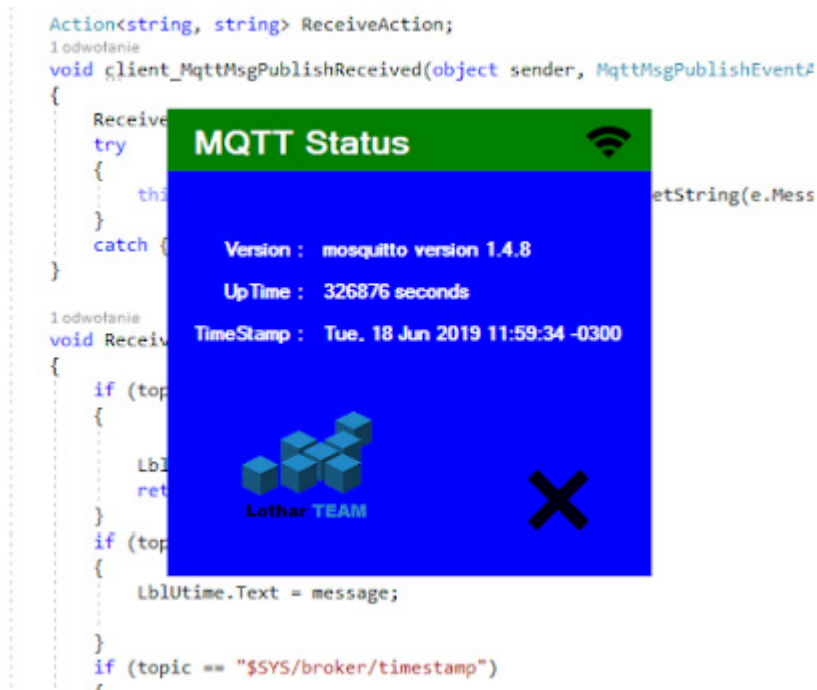
```
$>mosquitto_sub -t '$SYS/broker/timestamp' -u <nazwa> -P  
<hasło>
```

Tu możecie pobrać aktualną sygnaturę czasową, która może się okazać przydatna w wielu przypadkach, jeśli będzie trzeba opatrzyć dane właśnie parametrem timestamp. Oczywiście możemy wszystko ładnie podpatrywać w MQTT Explorerze. Jest tam sporo danych, które mogą się nam przydać na wiele sposobów, choćby do tworzenia statystyk działania serwera, liczby klientów czy ilości wysłanych i odebranych informacji, co warto mieć na uwadze (rysunek 12.6).



Rysunek 12.6. Widok na dane systemowe brokera

Dane te możemy np. odczytywać przez naszą aplikację monitorującą status brokera, który sobie gdzieś tam działa i którego nawet nie widać, bez konieczności logowania się do niego (rysunek 12.7).



Rysunek 12.7. Aplikacja SunMQTT Stats

Ten prosty programik znajdziecie na moim GitHubie pod adresem https://github.com/SunDUINO/SunMQTT_Stats wraz ze źródłami. Nie jest to bowiem miejsce na omawianie tego, jak takie małe чудо napisać, choć może kiedyś napiszę coś o tworzeniu aplikacji desktopowych.

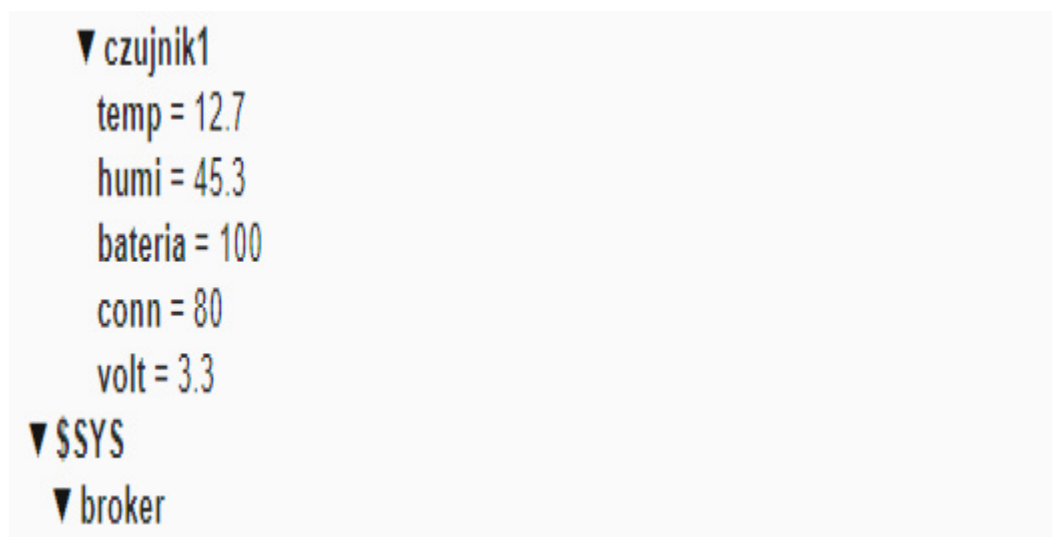
JSON — jak go zrozumieć

Postawiliśmy nasz broker na małym SBC klasy Raspberry Pi i już wiecie, jak podglądać dane. Nasze proste urządzenia będą przekazywać proste dane. Ale większość firmowych urządzeń używa struktury danych JSON i w sumie sam musiałem ją zrozumieć, by uniknąć problemów z czytaniem w aplikacjach klienckich tych informacji, które są mi potrzebne. Miałem z tym trochę kłopotów, więc postanowiłem skrobnąć dla Was to i owo na ten temat.

Dla przykładu, budujemy urządzenie, które wysyła następujące informacje:

- temperatura = 12,7,
- wilgotność = 45,3,
- stan baterii = 100,
- napięcie baterii = 3,3,
- jakość połączenia = 80%.

Jak widzicie, urządzenie będzie musiało pracować na zasilaniu bateryjnym, dlatego dane będą wysyłane tylko wtedy, gdy zostaną zmienione, trzeba bowiem oszczędzać energię... Najprościej będzie każdą informację wysłać osobno. Możemy je prezentować w osobnych tematach, które łatwo subskrybować w innych urządzeniach czy panelach (rysunek 12.8).



Rysunek 12.8. Temat czujnik1 i dostarczone dane

Jak widać na rysunku 12.8, jest to proste i skuteczne oraz łatwo czytać dane, ponieważ nasze urządzenie tylko subskrybuje osobne tematy, np. `/czujnik1/humi`. Ale z czasem natłok informacji robi się mniej wygodny i może przytłoczyć. Właśnie dlatego warto skorzystać z

dobrodziejstw JSON-a, bo i wygląda to dobrze, i jest czytelnie oraz oszczędnie... Wszystkie dane wysyłamy w jednej ramce:

```
msg = { }
```

JSON używa zazwyczaj pary znanej jako klucz-wartość.

Oznacza to, że klucz opisuje typ informacji, a wartość zawiera właściwą informację. Dostęp do tych informacji uzyskujemy za pomocą notacji `msg.payload` i `msg.topic`:

```
msg = {  
  "payload": "<jakiś payload>",  
  "topic": "<jakiś topic>"  
};
```

Fajne jest to, że taki obiekt może przechowywać nieograniczoną liczbę kluczy i wartości, które są oddzielone od siebie przecinkiem (,). Oczywiście nazwy kluczy muszą być unikalne i nie powinny zawierać spacji. Dzięki temu nasze dane z czujnika będą wyglądać np. tak jak na rysunku 12.9 (urządzenie mojego pomysłu, nazwane SoTToN_H&T).



```
► bridge (8 topics, 125 messages)  
► SoTToN SW2CH (1 topic, 1 message)  
  SoTToN_H&T = {"battery":45,"humidity":59.22,"linkquality":44,"temperature":29.4,"voltage":2600}  
▼ wifi
```

Rysunek 12.9. Przykład ramki danych w formacie JSON

Jak widać, klucze i przypisane im wartości są oddzielone dwukropkiem (:), a klucze są ujęte w cudzysłów (""), w przeciwieństwie do wartości. JSON może też przechowywać wiele formatów danych, co jest wygodne i wygląda tak:

- null,

- whitespace (),
- value (string, integer, bool null),
- object (zagnieżdżony JSON, ale zgodny z regułami JSON),
- boolean (true/false),
- string (wartości muszą być ujęte w ""),
- number (integer lub float, nie wymagają ""),
- array (uporządkowane wartości).

JSON — czytanie danych

Nieco już poznaliście Dżejsona. Wiecie, jakie dane może przekazywać i jak są one zorganizowane oraz jak wielką wygodą są dla naszych urządzeń IoT. W tym miejscu na przykładzie naszego czujnika zobaczycie, jak łatwo pobierać potrzebne informacje. Oczywiście nie wyczerpię tematu, co najwyżej lekko go poruszę, na tyle, na ile w tej chwili jest to nam przydatne.

Ale wróćmy do rzeczy...

Najfajniejsza sprawa w JSON-ie to łatwość, z jaką można uzyskać dostęp do uporządkowanych danych. Weźmy przykład naszego małego czujnika SoTToN_H&T, w którym jak pamiętacie, dane prezentują się następująco:

```
SoTToN_H&T =
{"battery":77,"humidity":48.48,"linkquality":44,"temperature":22.62,"voltage":2800}
```

Dane są uporządkowane w ramce zgodnej z zasadami struktury JSON. Możemy prosto wyczytać wszystkie informacje:

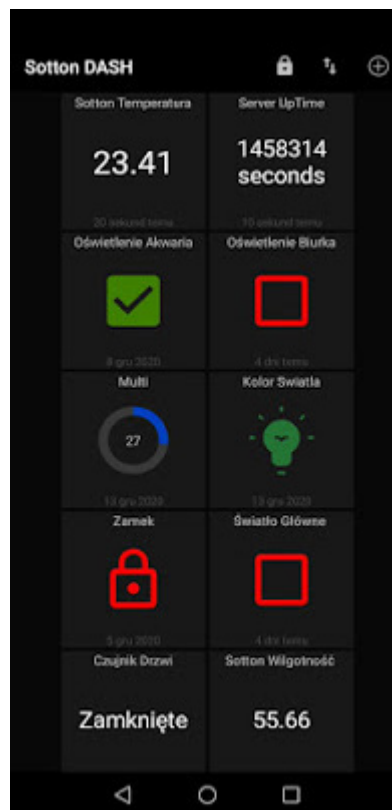
- stan baterii — 77%,
- wilgotność — 48,48%,
- jakość połączenia z siecią ZigBEE — 44%,
- temperatura — 22,62 stopnia Celsjusza,

- napięcie baterii — 2,8 V.

Widzicie? Mamy wszystko, co trzeba, ale chcemy prezentować dane w nieco ładniejszej, bardziej intuicyjnej formie, tak by łatwo było zrozumieć, co się dzieje. Dlatego zamiast zaglądać do MQTT Explorera, chcemy sprawdzić stan np. przez aplikację w telefonie. Założmy tym momencie, że interesuje nas tylko:

- wilgotność,
- temperatura.

Dla wygody umieścimy je w kafelkach o nazwach *Sotton Temperatura* i *Sotton Wilgotność*. W przykładzie jeden znajduje się u góry po lewej, a drugi po prawej na dole. W ten sposób prezentacja jest bardziej czytelna. Widać też kilka innych urządzeń, ale to na razie nie jest istotne (rysunek 12.10).



Rysunek 12.10. Przykładowe okno dashboarda na smartfonie

(W tej chwili używam MQTT Dashboarda, do którego jeszcze wrócimy).

By więc dostać się do danych, np. wilgotności, musimy podać temat, w którym są przechowywane... W tym przypadku jest to:

Sun/SoTToN_H&T

Następnie musimy wybrać, zgodnie z naszym przykładem, właściwy klucz... Chcemy czytać wilgotność, a więc kluczem jest tu humidity, co odczytujemy jako:

`$.humidity`

W następnym obiekcie czytamy wartość innego klucza.

Oczywiście można prezentować wiele informacji w jednym obiekcie/kafelku, przez rozdzielenie kluczy kropkami, np:

`$.humidity.temperature.voltage`

Prawda, że prosto i przyjemnie?

CZĘŚĆ IV. Do dzieła!

Wreszcie dotarliśmy do tego magicznego momentu, w którym, niczym alchemicy elektroniki, używamy różnych dostępnych na rynku składników, aby złączyć je w harmonijną sieć. Nasza kreatywność rozkwita niczym kwiaty na drzewach sakury, podobnie jak pole sprzętowe, na którym mamy dostęp do bogactwa modułów, płytek i czujników, a także swobodę w kreowaniu kompleksowych projektów. "Własne projekty" to szerokie pojęcie, obejmujące nawet te, które powstają z gotowych modułów, ale prawdziwą ekstazę dają te, które rozwijają się od abstrakcyjnej idei do rzeczywistego, działającego sprzętu opartego na własnej, zindywidualizowanej płytce.

Nie tak dawno temu projektowanie płytek było jak magia, wymagająca użycia parafiny, farby spirytusowej i kwasu solnego. Początki to było mozolne łączenie drucikami elementów na kawałku bakelitu. Potem nadeszła era termotransferu i B237, otwierając przed nami drzwi do dość powtarzalnych wyników. Jednak tworzenie dwustronnych płyt tą metodą było wyzwaniem. Produkcja prototypów w małych ilościach w prawdziwej płytkarni była kosztowna. Ale od czasu, gdy można tanio wykonać krótkie serie płytek, o różnych kolorach, kształtach, z frezowaniem i nawet montażem SMD, cały ten proces stał się znacznie łatwiejszy. Dzięki temu urządzenia elektroniczne tworzone przez amatorów weszły na wyższy poziom.

Teraz chciałbym Wam pokazać, jak zanurzyć się w projekcie, jak zaplanować i stworzyć własne urządzenie, jak je oprogramować. W tej podróży stworzymy swoistą enklawę — fabrykę elektronicznych marzeń konstruktora.

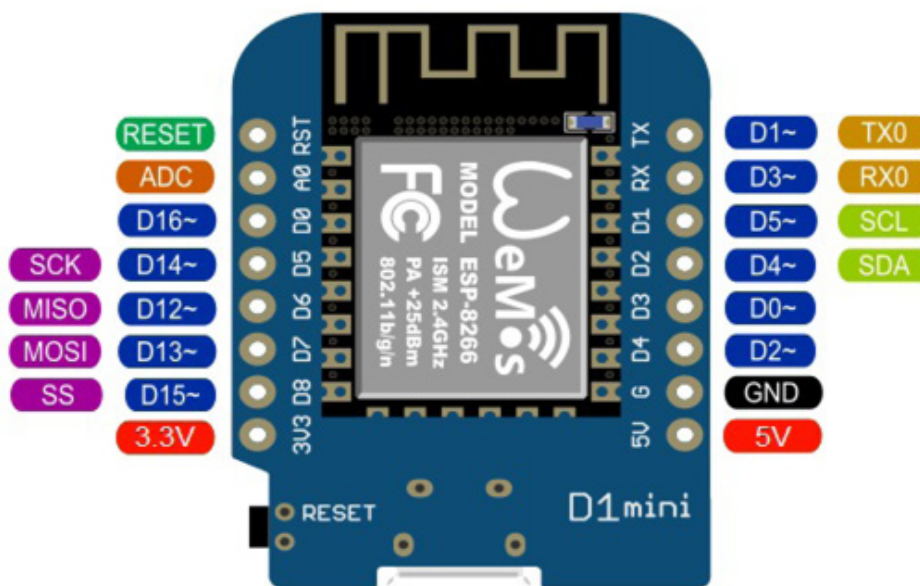
Tu warto poznać programy do projektowania obwodów elektronicznych, takie jak Eagle i KiCad, dzięki czemu łatwo i tanio stworzymy nasze profesjonalne urządzenia. Niemniej czy będziecie używać gotowych modułów, korzystać z uniwersalnych obwodów, czy projektować własne płytki, pamiętajcie, że każda droga prowadzi do celu, a z praktyką można osiągnąć niezwykle efekty — i to jeszcze w przystępny sposób.

13. Pierwszy projekt

W pierwszym projekcie oprzemy się na gotowcach, które łatwo zdobyć w rozsądnej cenie. Tu użyjemy taniej konstrukcji, powszechnie dostępnej w internecie, czy to w rodzimej witrynie aukcyjnej, czy też na znanej stronie zza Wielkiego Muru: niewielkiej płytki o wdzięcznej nazwie D1 mini. Ma na pokładzie układ CH340G, pozwalający wygodnie programować bez użycia programatora przez gniazdo USB. Sercem D1 mini oczywiście jest układ ESP8266 w postaci modułu ESP-12F, który ma 4 MB pamięci flash oraz, co dla nas ważniejsze:

- 11 cyfrowych wejść/wyjść,
- 10 kanałów PWM,
- 1 wejście analogowe.

Zapewnia też interfejsy szeregowy, takie jak UART, SPI, I2C, 1-Wire. To dość dużo jak na takie maleństwo, ale to dobrze, gdyż idealnie nadaje się do naszych niecnym planów stworzenia czujnika temperatury i wilgotności. W tym momencie, gdy już mamy płytkę, przyda nam się pinout. Opisy wyprowadzeń łatwo znaleźć w sieci, ponieważ udostępniają je sami dostawcy sprzętu. Tu na nasze potrzeby odpowiedni będzie pinout pokazany na rysunku 13.1.



Rysunek 13.1. Opis pinów płytki Wemos D1 mini¹

Jak zapewne zauważyliście, mamy tu wiele przydatnych dla nas informacji. Wszystko jest tu ładnie podane i łatwe w interpretacji dzięki informacjom o nazwach pinów i sygnałów. Dzięki temu możemy łatwo i szybko przystąpić do pracy.

Następną rzeczą, jaka się nam przyda, jest czujnik, np. popularny DHT11 lub DHT22, zwany też AM2302. Tu bardziej skupię się na DHT22, ma bowiem lepsze parametry w kontekście naszych zastosowań.

Choć wygląda moim zdaniem niepozornie, to jest tani i dosyć dobrze wywiązuje się ze swojego zadania. Oczywiście jest dostępnych sporo takich czujników, mniej lub bardziej zaawansowanych, o lepszych parametrach

i dokładniejszych. Nasz mały biały klocek wydaje się jednak wręcz idealny do naszego zastosowania. Charakteryzuje się następującymi parametrami:

- napięcie zasilania: od 3,3 V do 5,5 V,
- średni pobór prądu: 0,2 mA,
- temperatura:
 - zakres pomiarowy: od -40 do +80°C,
 - rozdzielczość: 8 bitów (0,1°C),
 - dokładność: $\pm 0,5^\circ\text{C}$,
 - czas odpowiedzi: średnio 2 s,
- wilgotność:
 - zakres pomiarowy: od 0 do 100% RH,
 - rozdzielczość: 8 bitów ($\pm 0,1\%$ RH),
 - dokładność: $\pm 2\%$ RH*,
 - czas odpowiedzi: średnio 2 s.

Jak więc widać, doskonale nada się nawet do powieszenia na balkonie. DHT22 na początek najlepiej kupić na gotowym module, co pozwoli łatwo go podłączyć z użyciem kilku przewodów — i już będzie gotowy do pracy. Pracy jest niewiele, gdyż DHT22 wymaga tylko trzech przewodów:

VCC — w naszym przypadku +3,3 V,

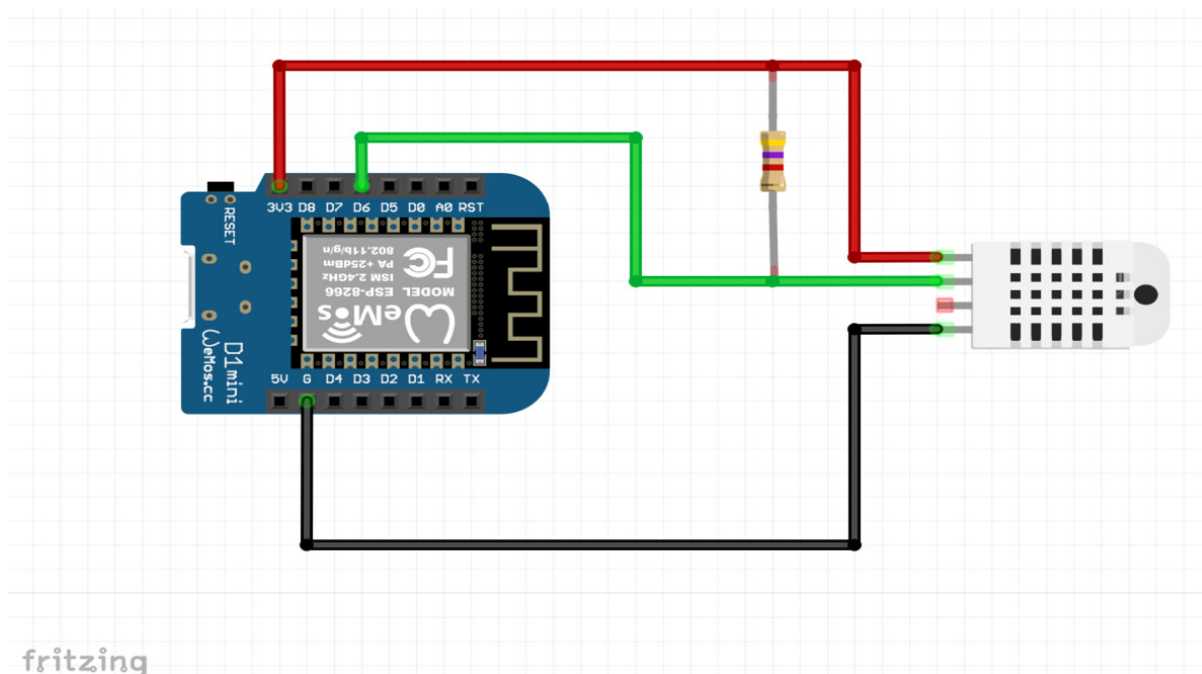
GND — masa,

DATA — cyfrowe wyjście czujnika, które podłączymy do jednego z pinów naszego ESP8266.

Możemy też użyć DHT11 w niebieskiej obudowie, który ma mniejszą dokładność i pracuje w zakresie temperatur od 0 do 50°C. Połączenia obu czujników są identyczne, ale do DHT11 przyda się rezystor 10 kiloomów, który będzie podciągał sygnał data do VCC, czyli, inaczej mówiąc, *pull-up*. W przypadku DHT22 ten rezystor nie jest konieczny.

Jak więc działa nasz DHT22? W skrócie wygląda to tak: wewnątrz modułu czujnika DHT22, jeśli byśmy wydobyli go z plastikowej obudowy, zobaczylibyśmy na ceramicznej płytce, pokrytej silikonowym zabezpieczeniem przed wpływem warunków atmosferycznych, termistor NTC. Jest to rezystor o ujemnym współczynniku temperatury, co oznacza, że rezystancja NTC spada wraz ze wzrostem temperatury, a rośnie z jej spadkiem. Jest tam też element wykrywający wilgotność — w zasadzie to kondensator, który zmienia swoją pojemność w zależności od ilości wilgoci w powietrzu. Z drugiej strony znajduje się mały mikrokontroler, który obsługuje gromadzenie, przetwarzanie i transmisję danych za pośrednictwem protokołu cyfrowego zgodnego z 1-Wire firmy Dallas (obecnie Maxim Integrated). Warto też wiedzieć, że DHT22 ma częstotliwość próbkowania 0,5 Hz, co oznacza, że aktualizacja danych temperatury i wilgotności następuje co 2 sekundy. Oczywiście możemy odczytywać układ co 0,5 sekundy i 1 sekundę, tylko że będziemy otrzymywać taki sam wynik. W sumie tak częste odpytywanie czujnika nie ma większego sensu i w większości przypadków wystarczy odczytywać pomiar co minutę lub nawet rzadziej.

Czas więc się podłączyć do płytki, co zwłaszcza w przypadku modułu DHT22 jest bardzo proste. Najlepszy będzie tu schemat narysowany w popularnym programie Fritzing (do którego jeszcze wrócimy). Jest pokazany na rysunku 13.2.



Rysunek 13.2. Przykład połączeń Wemod i DHT22²

Tak właśnie całość by się przedstawiała w razie użycia samego DHT11 lub DHT22. W przypadku DHT22, przypomnę, można pominąć rezystor; ja go czasem stosuję z przyzwyczajenia, w każdym razie jego obecność nie zaszkodzi. (Zdania na ten temat są podzielone). Uf, jakoś poszło, więc nie zwlekając, przejdźmy do najważniejszego etapu i tchnijmy nieco życia w wytwór naszej wyobraźni, a nic tego nie uczyni lepiej niż trochę prądu i program, który napiszemy. Właściwie to napiszemy trzy programy. W każdym z nich zastosujemy inną metodę prezentowania wyników pomiaru, tak żeby po drodze zobrazować, jak kod i jego różne warianty działają na naszym urządzeniu. Do programowania użyjemy platformy Arduino, gdyż w tak zaawansowanym projekcie, naszpikowanym kosmiczną technologią, nie ma sensu się wysilać.

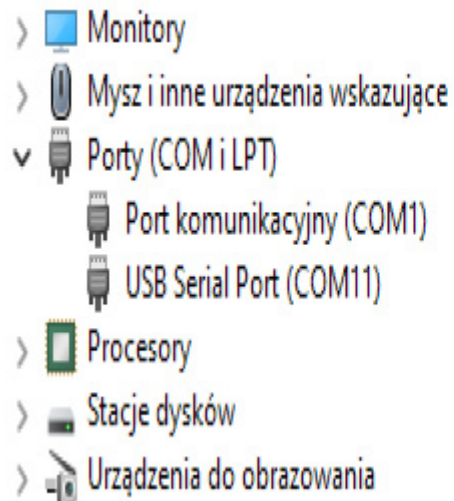
Oczywiście to najprostszy możliwy przypadek. Można by go jeszcze bardziej uprościć: wykonywać tylko odczyt

temperatury, ale moim skromnym zdaniem byłoby to nudne i w sumie podobne do migania diodą, a tak skonstruujemy zaawansowany czujnik temperatury i wilgotności, który może odgrywać pełnoprawną rolę pierwszego elementu naszego *smart home* oraz pierwszego urządzenia IoT.

Przebudzenie — wariant pierwszy

Pierwszym wariantem naszego przebudzenia będzie wyjątkowo prosty program. Jego działanie ograniczy się do sprawdzenia działania całego układu. Dane będą prezentowane w terminalu. Może mało to eleganckie, ale skuteczne i potwierdzi poprawność połączeń oraz pokaże działanie czujnika DHT22. W prosty sposób będziecie mogli sami sprawdzić, jak zmieniają się parametry, gdy np. chuchniecie na czujnik.

Podłączamy przewód USB do naszego frankensteina i czekamy na pioruny — może jednak lepiej, żeby nic nam nie wybuchło i nie iskrzyło. Upewnijmy się więc, że płytką D1 nie spoczywa na jakichś metalowych elementach, które mogłyby spowodować przypadkowe zwarcie. O ile strata D1 jest do przebolewania, o tyle uszkodzenie komputera lub samego portu USB może być kłopotliwe, a naprawa kosztowna (wiem o tym, sam bowiem na co dzień zajmuję się naprawami konsol i sprzętu komputerowego i często mam do czynienia z laptopami po dziwnych przygodach). W każdym razie ostrzegałem. Uf... nic się nie wydarzyło, wszystko zdaje się działać, zatem uruchamiamy wariant Arduino, który nam pasuje, i wybieramy (w sumie powinniśmy mieć wybraną) płytkę Generic ESP8266, a następnie ustawiamy właściwy port COM, pod którym zainstalował się układ CH430G. Dla porządku: u mnie jest to *COM11*, co widać w Menedżerze urządzeń (rysunek 13.3).



Rysunek 13.3. Menedżer urządzeń — porty COM

I teraz możemy przystąpić do napisania pierwszego programu prezentującego wynik w oknie terminala. Jak wspomniałem, program będzie bardzo prosty, więc nie będzie wodotrysków. W zasadzie cały wygląda tak:

```
#include <DHT.h>
#define DHTPIN 6
#define DHTTYPE DHT22
DHT dht(DHTPIN, DHTTYPE);
void setup()
{
    Serial.begin(115200);
    dht.begin();
}
void loop()
{
    delay(2000);
    float h = dht.readHumidity();
    float t = dht.readTemperature();
```

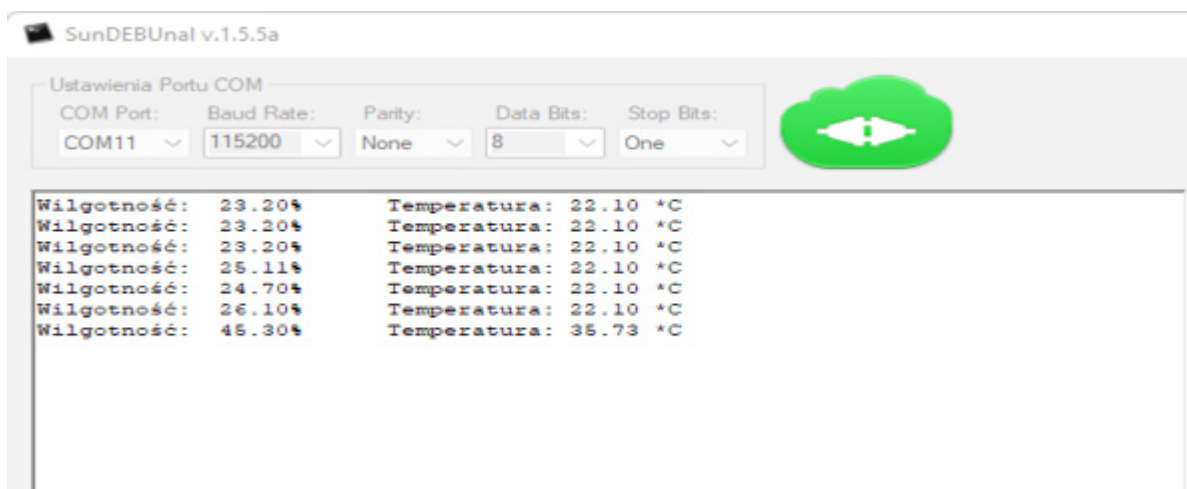
```

    Serial.print("Wilgotność: ");
    Serial.print(h);
    Serial.print("% Temperatura: ");
    Serial.print(t);
    Serial.print("°C ");
}

```

Nic szczególnego, ale liczy się działanie, a tu wynik jest poprawny, zgodnie z oczekiwaniem.

Ja użyłem własnego programu terminalowego (rysunek 13.4), ale może być to każdy inny, np. Smarty czy putty.



Rysunek 13.4. Okno terminala z danymi czujnika

Zmiany widoczne na rysunku 13.4 to efekt chuchania na czujnik. Tym sposobem ożywił się nasz układ. Potrafi on niewiele, a może właśnie całkiem sporo, ponieważ odczytuje temperaturę i wilgotność względną otoczenia. Sprawdziliśmy też przy okazji działanie naszego czujnika i połączeń. Teraz możecie się pobawić czujnikiem. A w międzyczasie omówimy pokrótce program, by wyjaśnić zasadę jego działania, gdyż celowo nie umieściłem w nim żadnego komentarza.

```
include <DHT.h>
```

W tym wierszu dodaliśmy do kodu bibliotekę obsługującą czujniki DHT, a dokładniej: DHT sensor library by Adafruit. Oszczędza nam ona całego zamieszania z obsługą czujnika. Biblioteka ta wymaga dwóch parametrów: numeru pinu *DATA* i typu czujnika. Żeby było łatwiej i czytelniej, definiujemy je w wierszach:

```
#define DHTPIN 6
```

```
#define DHTTYPE DHT22
```

Dowiadujemy się też stąd, że pin *DATA* podłączamy do pinu *GPIO6*; w razie potrzeby wystarczy zmienić numer pinu w tym właśnie wierszu. W drugim wierszu zdefiniowaliśmy typ czujnika jako *DHT22*. W razie potrzeby — lub gdy czujnik jest innego typu, np. *DHT11* — wystarczy zmienić ten wpis.

```
DHT dht(DHTPIN, DHTTYPE);
```

W tym wierszu informujemy bibliotekę, że nasz czujnik nazywa się *dht* i używa zdefiniowanych pinów i typu.

W funkcji *setup()* uruchamiamy port szeregowy i ustawiamy szybkość transmisji na 115 200 b/s oraz uruchamiamy magistralę dla czujnika *dht*. Następnie w pętli głównej *loop()* przechodzimy do realizacji odczytu i wysyłania na port szeregowy wyników pomiarów.

```
delay(2000) ; // Odczyt co 2 sekundy
```

```
float h = dht.readHumidity(); // Ustawiamy dla  
wilgotności zmiennoprzecinkową zmienną h, w której  
będą przechowywane dane odczytane z czujnika w postaci  
// ułamek dziesiętnego wyrażającego procentową wartość  
wilgotności względnej powietrza
```

```
float t = dht.readTemperature(); // Podobnie jak wyżej  
dla temperatury
```

```
// odczytanej z czujnika, wyrażonej w stopniach  
Celsjusza.
```

W następnych wierszach wysyłamy ciąg znaków zawierających nazwę i następnie odczytaną zawartość zmiennej przechowującej dane z czujnika. Efektem takiego zapisu jest ciąg odebrany przez terminal:

```
Wilgotność: 45.30% Temperatura: 35.73 *C
```

Dane są wyświetlane co 2 sekundy w kolejnych wierszach.

Jak więc widać, do prostych zastosowań warto używać prostych narzędzi, i w ten sposób szybko można zrealizować dany pomysł lub sprawdzić jego założenia i działanie. Nasz frankenstein działa prawidłowo, ale do IoT to mu daleko — niemniej coś działa. Oczywiście, żebyście nie marudzili, zrobimy na szybko to samo w SDK w PlatformIO i pod kontrolą RTOS. Potrzebna nam będzie biblioteka DHT, którą znajdziecie na moim GitHubie. Kod prezentuje się następująco:

```
#include <stdio.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <esp_err.h>
#include <dht/dht.h>

#define DHTPIN 6 // Definiujemy pin danych dla czujnika  
DHT, a więc GPIO6, czyli D6 na płytce D1

#define DHT_TYPE_DHT22 DHT22 // Definiujemy typ  
czujnika

void odczytDHT(void *arg)
{
    ESP_ERROR_CHECK(dht_init(DHTPIN, false)); //  
Sprawdzamy, czy DHT22 jest podłączony
    vTaskDelay(2000); // Oczekujemy 2 sekundy na pomiar
```

```

while (1)
{
    int wilgotnosc = 0;
    int temperatura = 0;

    if (dht_read_data(DHT22, DHTPIN, &wilgotnosc,
&temperatura) == ESP_OK)
    {
        // Wynik, jaki odczytamy z DHT, to: 604 =
60,4%, 252 = 25,2°C, więc wymaga
        // formatowania
        printf("Wilgotność: %d Temperatura: %d\n",
wilgotnosc, temperatura);
    }
    else {
        printf("Błąd odczytu czujnika DHT\n"); //
Wyświetlenie błędu w razie
// problemów z czujnikiem
    }
    vTaskDelay(5000);
}
vTaskDelete(NULL);
}

void app_main()
{
    xTaskCreate(odczytDHT, "odczytDHT", 2048, NULL,
tskIDLE_PRIORITY, NULL); // Uruchamiamy task odczytu
danych
}

```

Jak zatem widzicie, pisanie kodu pod kontrolą SDK jest równie proste i przyjemne. Kod jest bardzo podobny do wariantu napisanego w Arduino IDE; właściwie jedyna różnica polega na użyciu RTOS, dzięki czemu w tasku odczytujemy dane z czujnika *DHT22*. Jest to wygodne w przypadku większych programów, w których możemy mieć kłopot z funkcjami nieblokującymi. Wtedy z pomocą RTOS po prostu używamy tasków, a RTOS dba o resztę. Oczywiście to duże uproszczenie, ale mniej więcej o to chodzi. Cały program działa więc w taskach i realizuje te same funkcje co poprzedni. W kodzie zamieściłem komentarze, dlatego nie wymaga obszerniejszego omówienia.

Przebudzenie — wariant drugi

Było łatwo i prosto, ale niewiele miało to wspólnego z naszym zamysłem, więc teraz pójdziemy dalej i zbudujemy pierwsze urządzenie IoT. Nasz poprzedni czujnik rozbudujemy tak, że będzie się łączył z siecią WiFi i prezentował dane na własnym serwerze WWW. Będzie to urządzenie w pełni obsługiwane przez naszą małą płytkę D1 opartą na ESP8266. Od strony połączeń nic nie zmieniamy, całe zaś czary odbędą się w kodzie, który napiszemy. Dzięki temu zabiegowi będziemy mogli w prosty sposób odczytać parametry przez przeglądarkę internetową, np. za pomocą smartfona w sieci lokalnej, a gdy odpowiednio ustawimy router, IP naszego czujnika będzie dostępne w sieci zewnętrznej z dowolnego miejsca na ziemi. Czyli proste, ale jednak prawdziwe IoT. Czeką nas tu małe wyzwania, skoro bowiem mamy postawić serwer WWW na naszym małym ESP, musimy zrobić też małą stronkę w HTML, która będzie zawarta w kodzie naszego programu. Zarazem musimy umieścić na niej dane z naszego czujnika. No dobrze, przejdźmy do kodu.

```

#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>
#include <DHT.h>
// Dane do sieci Wi-Fi
const char *ssid = "<twoja sieć WiFi>";
const char *password = "<hasło do WiFi>";
// PIN, na którym podłączony jest czujnik DHT22
#define DHTPIN 2
#define DHTTYPE DHT22
// Inicjalizacja obiektu DHT
DHT dht(DHT_PIN, DHTTYPE);
// Obiekt serwera
ESP8266WebServer server(80);
void handleRoot()
{
    // Odczyt danych z czujnika DHT22
    float h = dht.readHumidity();
    float t = dht.readTemperature();
    // Przygotowanie strony HTML
    String html = "<html><body>";
    html += "<h1>Dane z czujnika DHT22:</h1>";
    html += "<p>Temperatura: " + String(t) + " &#8451;";
    html += "</p>";
    html += "<p>Wilgotność: " + String(h) + " %</p>";
    html += "</body></html>";
    // Wysłanie odpowiedzi do klienta
    server.send(200, "text/html", html);
}

```

```

void setup()
{
    // Inicjalizacja portu szeregowego
    Serial.begin(115200);
    // Inicjalizacja Wi-Fi
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(250);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println(WiFi.localIP()); // Wyświetla w
terminalu adres IP
    // Inicjalizacja czujnika DHT22
    dht.begin();
    // Konfiguracja obsługi ścieżki głównej ("/")
    server.on("/", HTTP_GET, handleRoot);
    // Uruchomienie serwera WWW
    server.begin();
    Serial.println("HTTP server started");
}

void loop()
{
    // Obsługa żądań klientów
    server.handleClient();
}

```

Kod mamy napisany. Jak widzicie, w zasadzie użyliśmy poprzedniego kodu, ale znacznie go rozwinęliśmy. Doszły też dwie nowe biblioteki:

```
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>
```

Pierwsza pozwala na nawiązanie połączenia WiFi i jego obsługę, a druga zapewnia obsługę serwera WWW i HTML. Jako że większość kodu już znacie, skupię się na opisie nowych elementów programu. Oczywiście całość pokrótce omówimy, ale już teraz każdy będzie wiedział, o co chodzi.

Te dwa wiersze:

```
const char *ssid = "Twoja_Siec_WiFi";
const char *password = "Haslo_Do_WiFi";
```

pozwalają na wprowadzenie nazwy i hasła WiFi, co pozwoli nam dołączyć do sieci i korzystać z jej zasobów, jeśli będzie to nam potrzebne.

W wierszu:

```
ESP8266WebServer server(80);
```

tworzymy obiekt server, który będzie reprezentował serwer HTTP na porcie 80. Czyli dokładnie tak, jak działają duże serwery.

W funkcji:

```
void handleRoot()
{
    float h = dht.readHumidity();
    float t = dht.readTemperature();
    String html = "<html><body>";
    html += "<h1>Dane z czujnika DHT22:</h1>";
    html += "<p>Temperatura: " + String(t) + " &#8451;";
    html += "</p>";
}
```

```
    html += "<p>Wilgotność: " + String(h) + " %</p>";  
    html += "</body></html>";  
    server.send(200, "text/html", html);  
}
```

wykonujemy trzy działania:

- Zajmujemy się odczytem czujnika DHT22 w znany już Wam sposób.
- Tworzymy stronę w HTML, która jest tu dość prosta, ale zawiera wszystkie potrzebne nam dane. Nasza strona jest w tym przypadku ciągiem znaków `String`. Zajmuje sporo miejsca w pamięci, ale na razie nie jest to dla nas istotne. Ważne jest to, że teraz widzicie, jak budować stronę HTML w kodzie programu.
- Ostatni wiersz wysyła odpowiedź z kodem 200, czyli "okej", i zawartością naszego HTML.

W funkcji `setup()` tradycyjnie ustawiamy kilka parametrów, takich jak port szeregowy, na którym za pomocą terminala będziemy mogli obserwować proces łączenia z siecią WiFi, jednocześnie obrazując poprawność połączenia lub jego brak w terminalu. Czasem w krytycznych miejscach programu użycie informacji wyświetlanej na terminalu poprzez `Serial.println()` może nas uchronić od wielu godzin poszukiwania błędu działania programu. Jest to swego rodzaju wygodna metoda debugowania kodu. Następnie uruchamiamy odczyt z czujnika DHT i w liniijkach:

```
server.on("/", HTTP_GET, handleRoot);  
server.begin();  
Serial.println("HTTP server started");
```

konfigurujemy obsługę ścieżki głównej serwera ("/") za pomocą funkcji `handleRoot()`, konieczną do poprawnego

wyświetlania naszej strony. Następny wiersz uruchamia serwer HTTP na porcie 80.

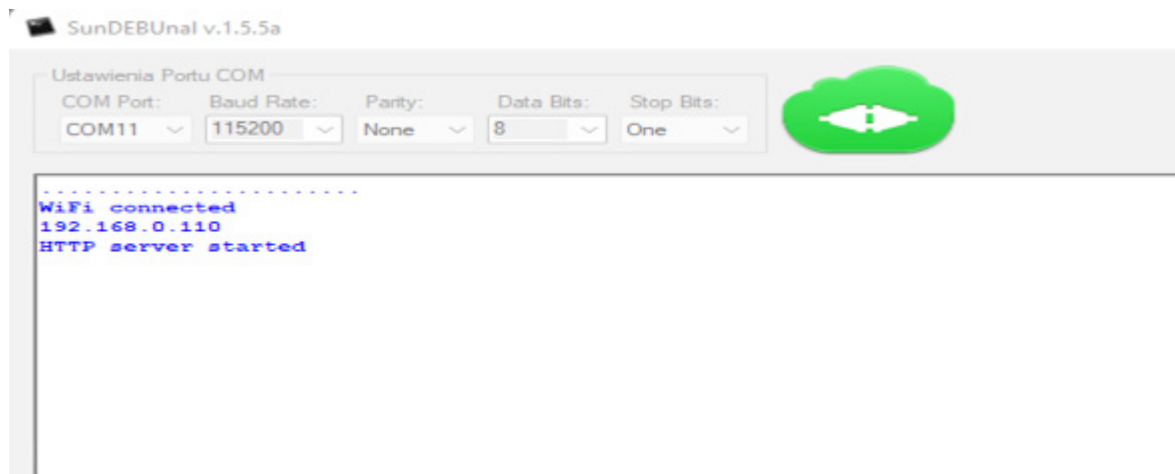
Kolejna linijka wysyła na terminal informację, że serwer został uruchomiony poprawnie.

W pętli głównej zaś:

```
void loop()
{
    server.handleClient();
}
```

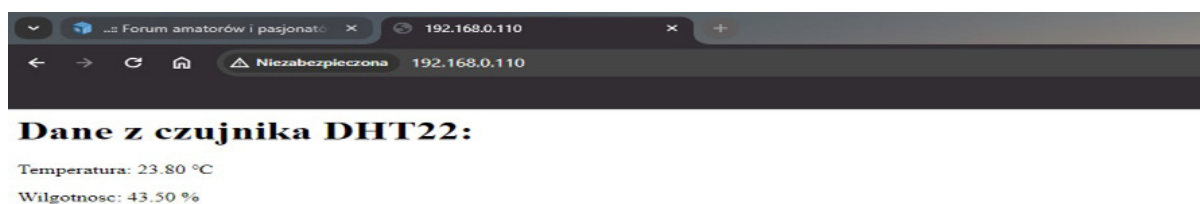
umieszczona została funkcja zajmująca się obsługą żądań klientów, czyli każdego urządzenia, które połączy się z serwerem poprzez przeglądarkę internetową, np. Chrome, po wpisaniu adresu. Ta funkcja umożliwi serwerowi reagowanie na przychodzące żądanie.

Jak widać, jest nieco zamieszania i nowych elementów, ale nie ma nic, czego nie da się łatwo ogarnąć. Szybko się okazuje, że to prosta sprawa i łatwa receptura na stworzenie prawdziwego urządzenia IoT, które łączy się zgodnie z zasadami z siecią WiFi i przekazuje dane z czujnika. Oczywiście, choć nasze maleństwo już potrafi zdalnie monitorować temperaturę i wilgotność pomieszczenia i jest gotowe do działania, to dalej jest prostym, niekoniecznie wygodnym w użyciu rozwiązaniem. To dlatego, że musimy się każdorazowo łączyć z czujnikiem przez przeglądarkę WWW. Niemniej urządzenie działa poprawnie i sprawnie rejestruje zmiany parametrów wraz z ich udostępnieniem w sieci (rysunek 13.5).



Rysunek 13.5. Okno terminala z adresem czujnika

Dzięki informacjom wysyłanym na terminal widzimy, że połączenie z naszą siecią WiFi się powiodło, a nasz serwer jest dostępny pod adresem *191.168.0.110*. Teraz wystarczy wpisać ten adres w przeglądarce, czy to smartfona, czy komputera, by zobaczyć dane odczytane z czujnika (rysunek 13.6).



Rysunek 13.6. Strona WWW czujnika w przeglądarce

Jak widać, wszystko funkcjonuje prawidłowo, uzyskane odczyty są poprawne. Program udostępnia nam jednak tylko podstawową funkcję monitorowania temperatury i

wilgotności za pomocą strony WWW. Przejdziemy więc do trzeciego wariantu, w którym rozbudujemy kod o możliwość korzystania z brokera MQTT.

Przebudzenie — wariant trzeci

Teraz napiszemy program, który będzie odczytywał dane z czujnika i wysyłał je do serwera MQTT, co umożliwi nam reagowanie na zmiany parametrów, np. automatyczne włączanie ogrzewania przy spadku temperatury. To już brzmi poważnie i zakrawa na skomplikowaną automatykę domową znaną z filmów science fiction. Tak, to możliwe, ale jak się zaraz przekonacie, jest stosunkowo łatwe w realizacji, a możliwości właściwie są ograniczone wyłącznie naszą wyobraźnią i zasobami sprzętowymi.

Jak pamiętacie, mamy na jakimś SBC uruchomiony broker MQTT Mosquitto. To dobra okazja, by go wykorzystać do odbierania danych z naszego czujnika, a w późniejszym czasie — również innych urządzeń. W tym celu musimy doinstalować sobie bibliotekę PubSubClient. Biblioteka ta pozwoli nam na publikowanie i subskrybowanie informacji z kanałów naszego brokera. Dlaczego jest to istotne? Załóżmy sobie taki oto prosty scenariusz.

Chcemy zautomatyzować i zoptymalizować ogrzewanie np. garażu. W tym przypadku nasz czujnik będzie stale monitorował temperaturę w garażu, a gdy ta spadnie poniżej danej wartości, automatycznie uruchomi się ogrzewanie, które się wyłączy, gdy temperatura osiągnie zadaną wartość. Będą nam potrzebne dwa urządzenia, z których jedno właśnie za chwilę oprogramujemy. Opierając się na naszym małym potworku Frankensteina, zmodyfikujemy kod:

```
#include <ESP8266WiFi.h>
```

```

#include <PubSubClient.h>
#include <DHT.h>

const char* ssid = "<twoja sieć WiFi>";
const char* password = "<hasło do WiFi>";
const char* mqtt_server = "<adres brokera MQTT>";
const char* mqtt_topic = "<twój temat MQTT>"; // np.
Czujnik1

#define DHTPIN 6
#define DHTTYPE DHT22
DHT dht(DHTPIN, DHTTYPE);

WiFiClient espClient;
PubSubClient client(espClient);

void reconnect()
{
    while (!client.connected())
    {
        Serial.print("Łączenie z MQTT brokerem...");
        if (client.connect("ESP8266Client"))
        {
            Serial.println("połączony");
        }
        else
        {
            Serial.print("nieudane, rc=");

```

```
        Serial.print(client.state());  
        Serial.println(" ponowna próba za 5  
sekund");  
        delay(5000);  
    }  
}  
}
```

```
void setup()  
{  
    Serial.begin(115200);  
    WiFi.begin(ssid, password);  
    while (WiFi.status() != WL_CONNECTED)  
    {  
        delay(250);  
        Serial.print(".");  
    }  
    Serial.println("");  
    Serial.println("WiFi Połączono");  
  
    dht.begin();  
  
    client.setServer(mqtt_server, 1883);  
    client.setCallback(callback);  
}
```

```
void loop()  
{
```

```

    if (!client.connected())
    {
        reconnect();
    }
    client.loop();

    float h = dht.readHumidity();
    float t = dht.readTemperature();

    Serial.print("Temperatura: ");
    Serial.print(t);
    Serial.print(" °C, Wilgotność: ");
    Serial.print(h);
    Serial.println(" %");

    String temperatura = String(t);
    String wilgotnosc = String(h);

    // Publikowanie danych w temacie MQTT
    client.publish((String(mqtt_topic) +
"/temperatura").c_str(), temperatura.c_str());
    client.publish((String(mqtt_topic) +
"/wilgotnosc").c_str(), wilgotnosc.c_str());

    delay(5000); // Odczyt i publikacja co 5 sekund
}

```

Tak by wyglądał nasz zmodyfikowany kod. Jak można zauważyć, niewiele się różni od poprzednich wersji, ale tym razem umożliwia wysłanie danych do brokera. Nasz broker

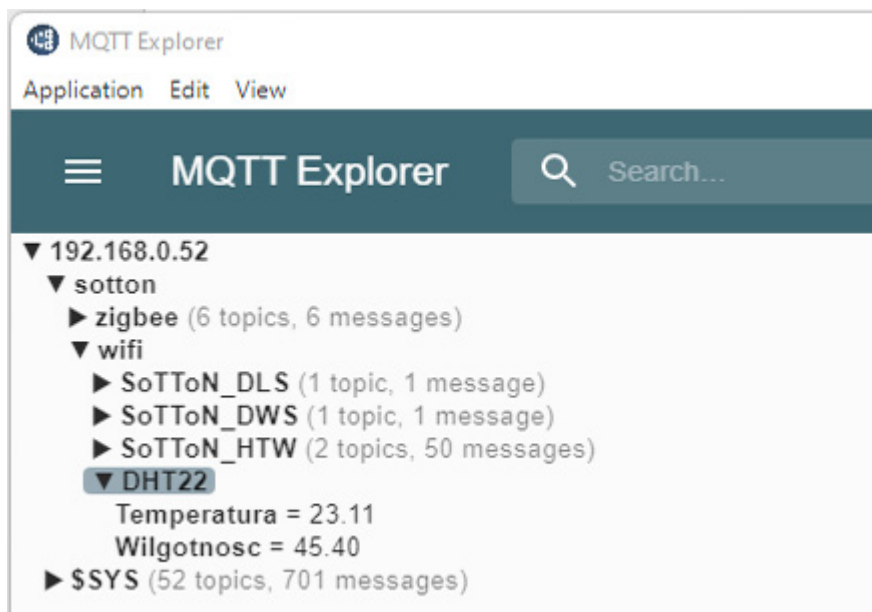
tymczasowo pracuje na domyślnych wartościach, więc nie musimy podawać do niego hasła. Wystarczy w linijce:

```
const char* mqtt_server = "<adres brokera MQTT>";
```

wpisać adres IP naszego serwera. Kolejną rzeczą jest ustalenie tematu dla urządzenia, co robimy w linijce:

```
const char* mqtt_topic = "<twój temat MQTT>";
```

Jeśli skompilujemy program w obecnej postaci, to pojawi się nam temat *<twój temat MQTT>*, a w nim dane, które nas interesują. Dla poprawienia czytelności, gdy mamy wiele czujników, warto więc zmienić nazwę w tej linijce np. na *DHT22* lub, jeśli mamy więcej czujników, na inny, o nazwie pozwalającej łatwo sprecyzować, o który nam chodzi. W ten sposób otrzymamy czytelne dane, które w MQTT Explorerze prezentują się tak jak na rysunku 13.7.



Rysunek 13.7. Czujnik w MQTT Explorerze

Prawda, że to wygodne i łatwe do przeglądania? Oczywiście możemy te dane łatwo podłożyć do jakiegoś programu typu MQTT Dashboard, dzięki czemu zyskamy ich łatwy podgląd

w każdym momencie, a jeśli nasz broker MQTT ma ustawiony na routerze dostęp do sieci zewnętrznej, to działa w zasadzie w każdym miejscu na ziemi, bo wystarczy tylko dostęp do sieci. W naszym prostym programie omówienia wymaga w sumie tylko funkcja `reconnect()`. Jest ona dość ważna, gdyż obsługuje połączenie z brokerem w razie rozłączenia. Dzięki temu urządzenie nawet po awarii sieci WiFi jest w stanie odtworzyć połączenie automatycznie i udostępnić nam aktualne dane z czujnika.

Tym sposobem stworzyliśmy pierwsze urządzenie IoT, którym jest czujnik temperatury i wilgotności. Od tej pory mamy sprawnie działający fragment systemu domowej automatyki znanej powszechnie jako *smart home*. Czas na kolejne sterowniki i urządzenia monitorujące.

[1](#) Źródło: Schemat stworzony w narzędziu Fritzing; dostęp 11.06.2024.

[2](#) Źródło: Schemat stworzony w narzędziu Fritzing; dostęp 11.06.2024.

14. Sterowanie urządzeniami

Sterowanie urządzeniami, zwłaszcza tymi związanymi z elektroniką, to dziedzina, która coraz bardziej wkracza w nasze codzienne życie. Niezależnie od tego, czy pracujemy nad projektem DIY, czy korzystamy z zaawansowanych systemów automatyki domowej, ważne jest, aby zdawać sobie sprawę z potencjalnych zagrożeń związanych z wysokim napięciem. Zatem na samym początku, nim przejdziemy do realizacji, omówimy nie tylko to, jak skutecznie sterować urządzeniami, ale także to, dlaczego zawsze powinniśmy zachować ostrożność i przestrzegać zasad bezpieczeństwa.

Dlaczego bezpieczeństwo jest priorytetem?

Wysokie napięcie, które może występować w elektronicznych obwodach i urządzeniach, stanowi potencjalne zagrożenie dla zdrowia i życia. Bezpieczeństwo powinno być zawsze na pierwszym miejscu, zarówno dla hobbystów, jak i profesjonalistów. Nieprzestrzeganie zasad bezpieczeństwa może prowadzić do niebezpiecznych sytuacji, takich jak porażenie prądem, pożar czy uszkodzenie sprzętu.

Kilka zasad bezpieczeństwa przy sterowaniu urządzeniami

Izolacja elektryczna

Korzystaj z narzędzi i urządzeń z izolowanymi rękojeściami, a także stosuj odpowiednie obudowy i zabezpieczenia, by uniknąć porażenia prądem elektrycznym.

Przerwy w zasilaniu

Zawsze odłączaj zasilanie przed jakimkolwiek manipulowaniem przy urządzeniach. To proste działanie może znacznie zmniejszyć ryzyko porażenia prądem.

Znajomość urządzeń

Zanim zaczniesz pracować z danym urządzeniem, zapoznaj się z jego specyfikacją elektryczną. Sprawdź, jakie napięcia i prądy są obsługiwane, aby uniknąć przeciążenia.

Używaj odpowiednich narzędzi

Do każdego zadania wybieraj odpowiednie narzędzia. Nie używaj narzędzi, które nie są przeznaczone do pracy w danym środowisku.

Sterowanie urządzeniami za pomocą mikrokontrolerów

Współczesne projekty, nie tylko DIY, często wykorzystują mikrokontrolery, takie jak ESP8266, do sterowania różnymi urządzeniami. Te platformy oferują szeroki zakres możliwości, ale jednocześnie zwiększają ryzyko związane z manipulowaniem zasilaniem. Dlatego też przy sterowaniu urządzeniami za pomocą mikrokontrolerów ważne są zasady bezpieczeństwa, a także dodatkowe środki ostrożności. Oto kilka zaleceń:

Separacja zasilania

Zastosuj oddzielne źródła zasilania dla mikrokontrolera i dla urządzenia, którym steruje. To zabezpieczenie może

pomóc w uniknięciu ewentualnych zakłóceń i zminimalizowaniu ryzyka uszkodzeń sprzętu i porażeń.

Zabezpieczenia przeciwprzepięciowe

Wykorzystaj elementy takie jak przerywacze, bezpieczniki i inne zabezpieczenia przeciwprzepięciowe, aby chronić zarówno urządzenie, jak i siebie.

Ostrzeżenia przed wysokim napięciem

Umieść ostrzeżenia przed wysokim napięciem w miejscach, gdzie manipulujesz elektroniką i gdzie może występować napięcie sieciowe. To prosta, ale skuteczna metoda ostrzegawcza. Zauważ, że wszystkie urządzenia elektroniczne, w których może występować wysokie napięcie, czyli te zasilane z sieci energetycznej, są wyposażone w takie ostrzeżenia.

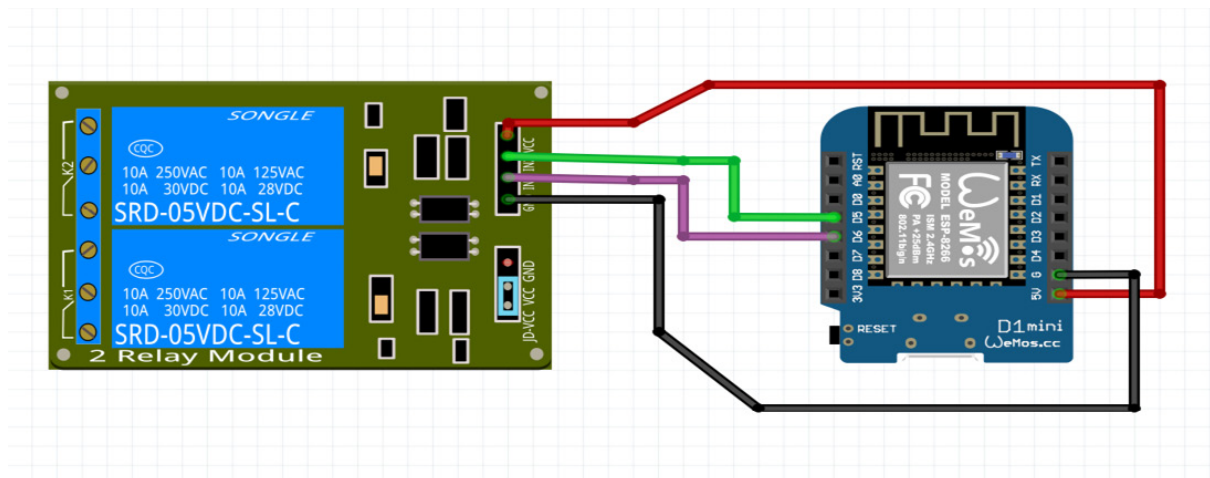
Podsumowanie

Przestrzeganie zasad BHP w pracy z urządzeniami zasilanymi wysokim napięciem lub przełączającymi wysokie napięcie jest kluczowe dla naszego zdrowia i bezpieczeństwa. Wysokie napięcia i prądy mogą bowiem powodować poważne obrażenia. Zarówno hobbysta, jak i profesjonalista powinni być świadomi ryzyka związanego z pracą nad elektronicznymi projektami. Pamiętajmy, że zawsze warto poświęcić dodatkowy czas na zapewnienie bezpieczeństwa w miejscu pracy.

Przestrzegajmy zasad bezpieczeństwa, stosujmy się do specyfikacji technicznych urządzeń, używajmy izolacji elektrycznej i bądźmy świadomi ryzyka. Tylko w ten sposób możemy cieszyć się swoimi projektami elektronicznymi, jednocześnie zachowując pełne bezpieczeństwo.

Schemat połączeń

Podobnie jak poprzednio użyjemy płytki Wemos D1 mini, ale może być to dowolna z układem ESP8266 czy ESP32. Dodatkowo przyda nam się tania płytka z dwoma przekaźnikami. Schemat połączeń narysujemy we Fritzingu, żeby sobie ułatwić ich obrazowanie, choć ja będę się tutaj posiłkował swoim projektem CircleBoard, opartym na płytce wyposażonej w ESP8266 i dwa przekaźniki. Na rysunku 14.1 przedstawiłem sposób podłączenia przekaźników do płytki Wemos D1 mini.



Rysunek 14.1. Podłączenie przekaźników do ESP8266³

Plusem tego rozwiązania jest to, że za nieduże pieniądze mamy w zasadzie szybkie efekty, gdyż dzięki gotowym modułom nie musimy zajmować się projektowaniem izolacji połączeń na płytce, a dodatkowo otrzymujemy separację optyczną pinów sterujących, całość sprowadza się więc do podłączenia czterech przewodów. W tym momencie nie ma tu dla nas niebezpiecznego napięcia. Będziemy też słyszeć w tym eksperymencie i widzieć efekt działania naszego kodu, ponieważ płytka przekaźników — poza tym, że

przełączniki charakterystycznie na niej klikają — ma diody LED obrazujące zasilanie modułu i włączenie poszczególnych kanałów (przełączników). Zarówno moduł jednokanałowy, jak i bardziej rozbudowany podłączamy dokładnie w ten sam sposób. Wiele tu zależy od potrzeb i liczby dostępnych wyjść mikrokontrolera. Powiedzmy jednak, że w tym przypadku dwa przełączniki są lepsze od jednego, i tej wersji się trzymamy. Zapytacie, gdzie to sterowanie... Otóż do wyjścia przełącznika podłączamy sterowane urządzenie.

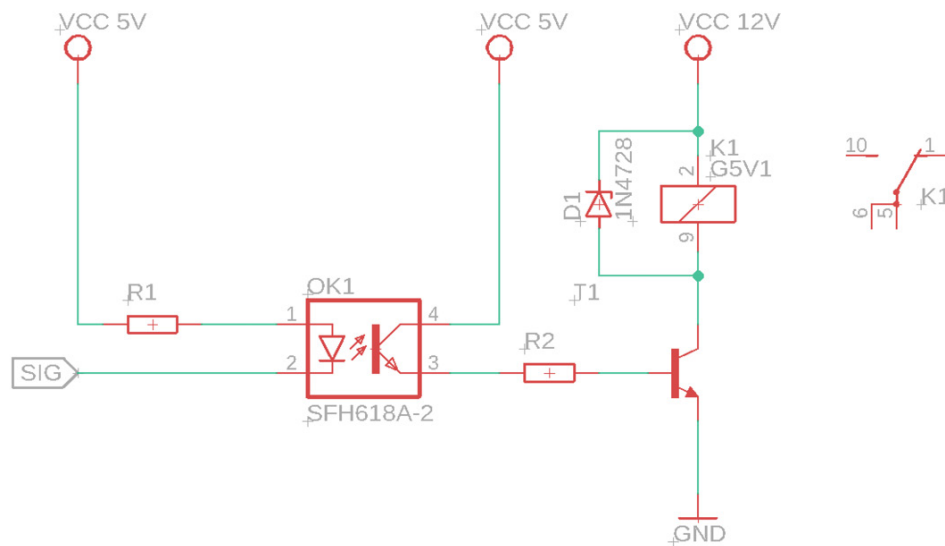
Pokrótkie wyjaśnię początkującym, że przełącznik to urządzenie elektryczne służące do sterowania innym urządzeniem lub obwodem elektrycznym. Jego główną funkcją jest przełączanie obwodu elektrycznego albo jego fragmentu w odpowiedzi na zmiany w innym obwodzie sterującym. Innymi słowy, to taki przełącznik, którym możemy sterować. Podstawowym składnikiem przełącznika jest cewka, zwykle nawinięta na rdzeniu ferromagnetycznym, oraz zestaw styków. Cewka jest podłączona do obwodu sterującego, w którym płynący prąd generuje pole magnetyczne, co powoduje przyciąganie lub odpychanie styków. Zmiana położenia styków w wyniku działania cewki prowadzi do otwarcia lub zamknięcia obwodu głównego.

Na stykach przełącznika nie musi tu wcale występować wysokie napięcie; sterować możemy np. listwami LED 12- lub 24-woltowymi. Niemniej miejmy na uwadze bezpieczeństwo.

Wracając do tematu — ja posłużę się swoją płytką, gdyż mam ją pod ręką i zawiera już dwa przełączniki, co też ułatwi mi pracę i pokazanie działania konstrukcji. Podobnie jak na początku użyłem jej w przypadku programu Blink. Tu warto też wspomnieć, że nasz układzik ESP nie jest przystosowany do sterowania bezpośrednio samym

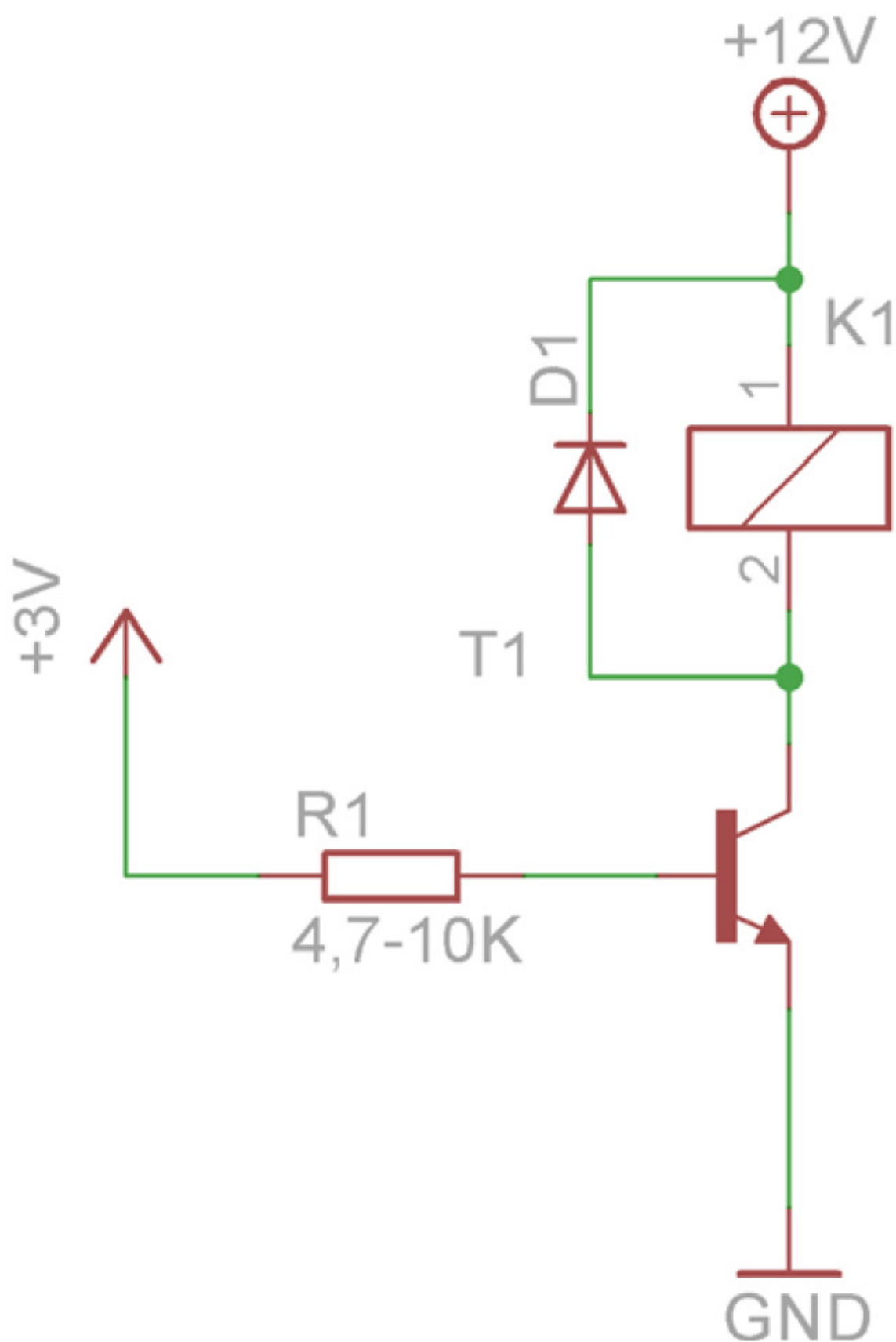
przełącznikiem. Dlatego polecam używanie gotowych modułów, które zawierają wszystkie potrzebne elementy. Na mojej płytce jest też zawarty tranzystorowy układ umożliwiający sterowanie przełącznikiem. Popatrzmy, jak to wygląda w praktyce:

1. Moduł, który Wam zaproponowałem, steruje przełącznikiem za pomocą tranzystora, ale wejście sygnału sterującego dodatkowo jest zabezpieczone transoptorem, który ponadto optycznie izoluje pin GPIO od układu z wyższym napięciem, co na schemacie wygląda jak na rysunku 14.2.



Rysunek 14.2. Schemat podłączenia przełącznika z transoptorem⁴

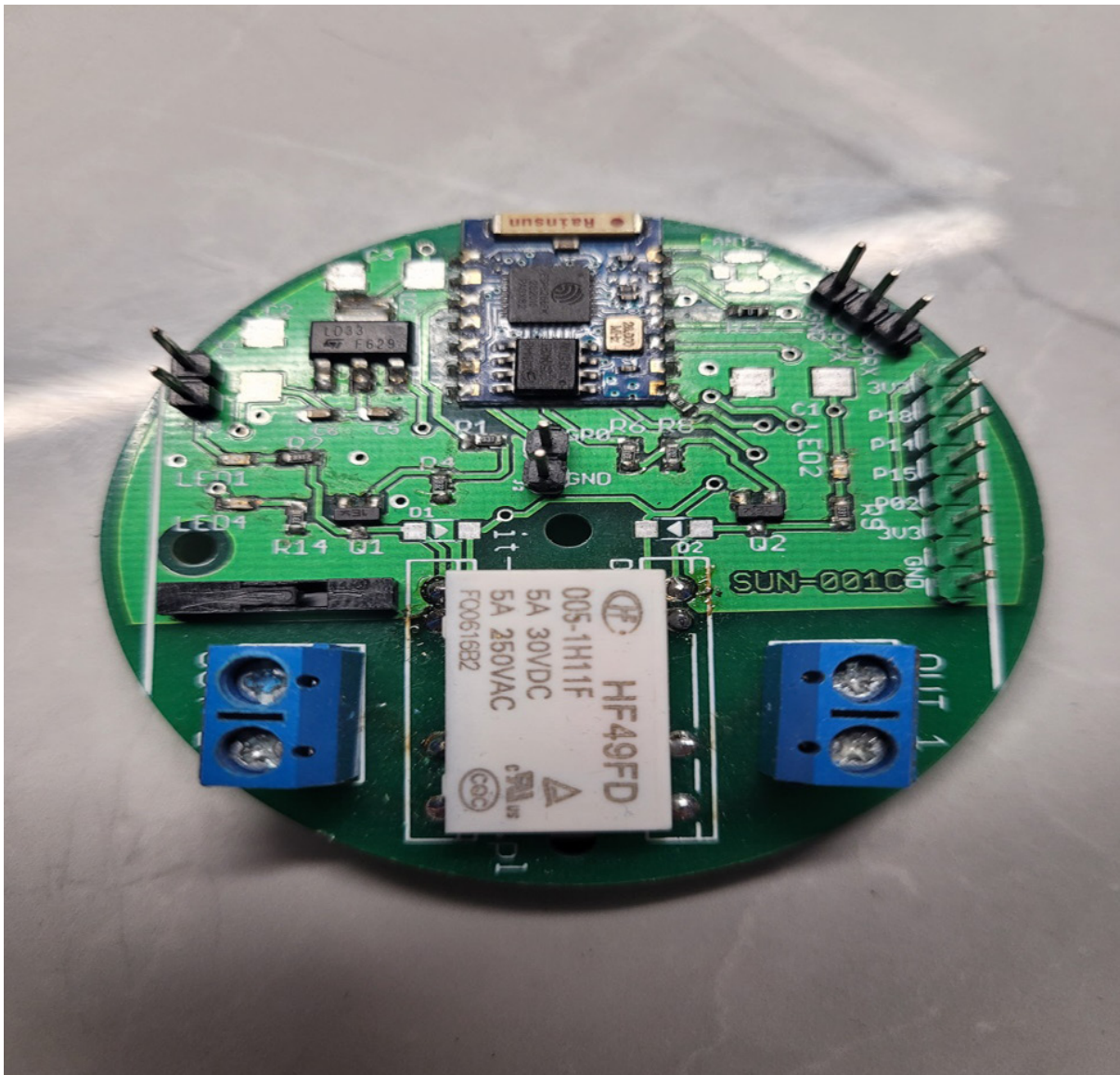
2. Moja płytka używa prostszej wersji sterowania, pozbawionej transoptora (rysunek 14.3).



Rysunek 14.3. Schemat podłączenia przekaźnika⁵

Obie wersje działają tak samo. Tranzystor, tu NPN, jest konieczny, gdyż przekaźnik jest przeznaczony do działania z wyższym napięciem i potrzebuje więcej prądu, niż jesteśmy w stanie pobrać z pinu ESP. Połączenie bezpośrednio spowodowałoby nieuchronnie uszkodzenie GPIO lub całego ESP. Na schematach widzimy napięcia 12 i 24 V; ja używam 5-woltowych przekaźników, co pozwala zasilать je z układu zasilania na płytce. ESP8266 i ESP32 działają, przypominę, na napięciu 3,3 V. Dioda przy przekaźniku to dioda prostownicza, odgrywająca tutaj rolę gasika, gdyż cewka przekaźnika jest elementem indukcyjnym, który w momencie wyłączenia może generować niepożądane skoki napięcia i tym samym uszkodzić inne elementy obwodu. Oczywiście są to podstawy podstaw, dlatego zakładam, że nawet ci początkujący mają je opanowane choćby w podstawowym stopniu, więc taki skrótowy opis powinien wystarczyć.

Wracając do meritum tematu — schematy połączeń mamy odhaczone. Płytkę, z którą będę pracował w tym rozdziale, jest zaś pokazana na rysunku 14.4.



Rysunek 14.4. CircleBoard

Krótko mówiąc, nie jest ona tak wygodna w użyciu jak D1, ale zawiera dwa przekaźniki, umieszczone po obu stronach płytki, dwie diody LED, a właściwie trzy, przy czym jedna jest od zasilania, i dodatkowo cztery piny GPIO, których można do czegoś użyć.

Na marginesie: w naszym scenariuszu z ogrzewaniem garażu całość możemy zawrzeć na tej jednej płytce. Tak,

pozwała nawet zbudować w pełni autonomiczne urządzenie kontrolujące temperaturę w pomieszczeniu, gdyż może samodzielnie odczytywać dane z czujnika DHT i samodzielnie podejmować decyzję o włączeniu ogrzewania. W uproszczeniu można by powiedzieć, że to taka miniaturowa sztuczna inteligencja.

Tymczasem mamy już wszystko gotowe do programowania.

Jeszcze przypomnienie o bezpieczeństwie: pamiętajcie, że przekaźniki są różne i mają różne parametry, nie tylko napięcie zasilania cewki, które w naszych projektach będzie oscylować głównie w zakresie od 5 do 12 V. Ważnym parametrem jest wytrzymałość styków. W przypadku przekaźników na gotowych modułach zwykle jest to 250 V, 10 A, przy $\cos \phi = 1$ (obciążenie rezystancyjne). Moje maluszki dopuszczają tylko 5 A przy 250 V. Możemy łatwo policzyć ze wzoru:

$$\text{moc (W)} = \text{napięcie (V)} \cdot \text{prąd (A)}$$

jakie najmocniejsze urządzenie możemy podłączyć do takiego przekaźnika. Policzmy dla mojego maluszka:

$$W = 230 \cdot 5$$

$$W = 1150$$

No, czajnik elektryczny teoretycznie uciągnie, ale to maksimum i raczej lepiej będzie podzielić je na pół i nie przekraczać właśnie połowy maksymalnej wartości. Ostatecznie oświetleniem możemy sterować bez problemu. W razie czego można będzie podłączyć do przekaźnika stycznik o znacznie większych możliwościach.

Oprogramowanie sterownika

Teraz możemy zająć się naszym programem. Poniekąd będzie on rozwinięciem wcześniejszego. Zasada działania też nie będzie specjalnie skomplikowana. Całość od strony klienta wygląda tak: program będzie przyjmował wartości ON i OFF. Jeśli zaś chodzi o stronę magiczną, ponownie będziemy po prostu łączyć się z siecią WiFi i naszym brokerem MQTT, gdzie w jednym temacie będziemy publikować stan kanału, a w drugim subskrybować wartości ON i OFF. Spowoduje to, że np. ze smartfona lub WallPanela będziemy wysyłać wiadomość ON lub OFF, a nasz układ będzie zmieniał wyjścia przekaźnika i publikował stan. Prawda, że proste? Zakładam, że już oczami wyobraźni widzicie, jak to będzie działało. Oczywiście nieco później zajmiemy się aplikacją sterująco-kontrolną, ale już teraz możemy snuć plany, jak będzie to wyglądało w całości i funkcjonowało. W tej chwili nie mniej ważny jest program dla naszego sterownika.

```
#include <ESP8266WiFi.h>
```

```
#include <PubSubClient.h>
```

```
// Ustawienia Wi-Fi
```

```
const char* ssid = "<twoja nazwa sieci>";
```

```
const char* password = "<twoje hasło sieci>";
```

```
// Ustawienia MQTT
```

```
const char* mqtt_server = "<twój adres IP serwera MQTT>";
```

```
const int    mqtt_port = 1883;
```

```
const char* mqtt_user = "<twoja nazwa użytkownika MQTT>";
```

```
const char* mqtt_password = "<twoje hasło MQTT>";
```

```
const char* mqtt_topic_pk1 = "włącznik/przekaznik1";
```

```
const char* mqtt_topic_pk2 = "włącznik/przekaznik2";
const char* mqtt_topic_s_pk1 =
"włącznik/przekaznik1/status";
const char* mqtt_topic_s_pk2 =
"włącznik/przekaznik2/status";
```

```
// Piny podłączenia przekaźników
```

```
const int przekaznik1 = 12; // Pin przekaźnika 1
```

```
const int przekaznik2 = 13; // Pin przekaźnika 2
```

```
// Inicjalizacja klienta Wi-Fi
```

```
WiFiClient espClient;
```

```
PubSubClient client(espClient);
```

```
// Funkcja obsługująca otrzymane wiadomości MQTT
```

```
void callback(char* topic, byte* payload, unsigned int
length)
```

```
{
```

```
    // Konwertuj payload na string
```

```
    String message = "";
```

```
    for (int i = 0; i < length; i++)
```

```
    {
```

```
        message += (char)payload[i];
```

```
    }
```

```
    // Sprawdź, czy otrzymano wiadomość dla przekaźnika
```

```
1
```

```
    if (String(topic) == mqtt_topic_pk1)
```

```
    {
```

```

        if (message == "ON")
        {
            digitalWrite(przekaznik1, HIGH); // Włącz
przekaznik 1
            client.publish(mqtt_topic_s_pk1, "ON");
        }
        else if (message == "OFF")
        {
            digitalWrite(przekaznik1, LOW); // Wyłącz
przekaznik 1
            client.publish(mqtt_topic_s_pk1, "OFF");
        }
    }

    // Sprawdź, czy otrzymano wiadomość dla przekaźnika
2
    else if (String(topic) == mqtt_topic_pk2)
    {
        if (message == "ON")
        {
            digitalWrite(przekaznik2, HIGH); // Włącz
przekaznik 2
            client.publish(mqtt_topic_s_pk2, "ON");
        }
        else if (message == "OFF")
        {
            digitalWrite(przekaznik2, LOW); // Wyłącz
przekaznik 2
            client.publish(mqtt_topic_s_pk2, "OFF");
        }
    }

```

```

        }
    }
}

void setup()
{
    Serial.begin(115200);
    // Inicjalizacja pinów przekaźników
    pinMode(przekaznik1, OUTPUT);
    pinMode(przekaznik2, OUTPUT);

    // Inicjalizacja połączenia WiFi
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.println("Łączenie z WiFi...");
    }
    Serial.println("Połączono");

    // Inicjalizacja klienta MQTT
    client.setServer(mqtt_server, mqtt_port);
    client.setCallback(callback);
}

void reconnect()
{
    // Pętla czekająca na ponowne połączenie z brokerem
    MQTT

```

```

while (!client.connected())
{
    Serial.println("Nawiazuje polaczenie z Brokerem
MQTT...");
    // Próba połączenia z klientem MQTT
    if (client.connect("ESP8266Client", mqtt_user,
mqtt_password))
    {
        Serial.println("Połączono.");
        // Subskrybuj tematy przekaźników
        client.subscribe(mqtt_topic_pk1);
        client.subscribe(mqtt_topic_pk2);
    }
    else
    {
        Serial.print("Bład, rc=");
        Serial.print(client.state());
        Serial.println(" Ponowna proba za 5
sek....");
        delay(5000);
    }
}

void loop() {
    // Sprawdź, czy klient MQTT jest połączony
    if (!client.connected())
    {
        reconnect();
    }
}

```

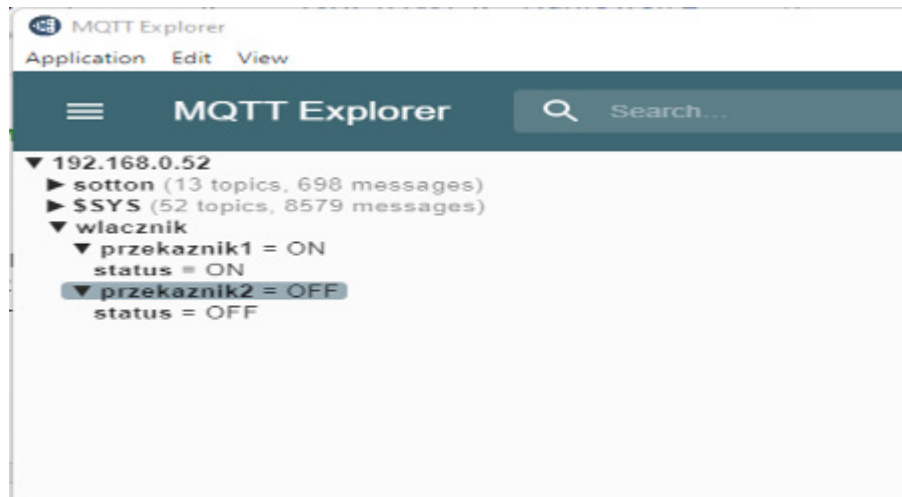
```
// Obsługa komunikacji MQTT
client.loop();
}
```

Kod wygląda nieźle, ale wprowadziliśmy tu kilka nowych elementów, gdyż o ile dobrze pamiętam, dla bezpieczeństwa ustawiliśmy też użytkownika i hasło dla naszego brokera. Dlatego w tym kodzie dodałem logowanie do MQTT. Zgodnie z założeniami możemy naszym sterownikiem sterować ze smartfona poprzez darmową aplikację MQTT Dashboard.

Program realizuje następujące zadania:

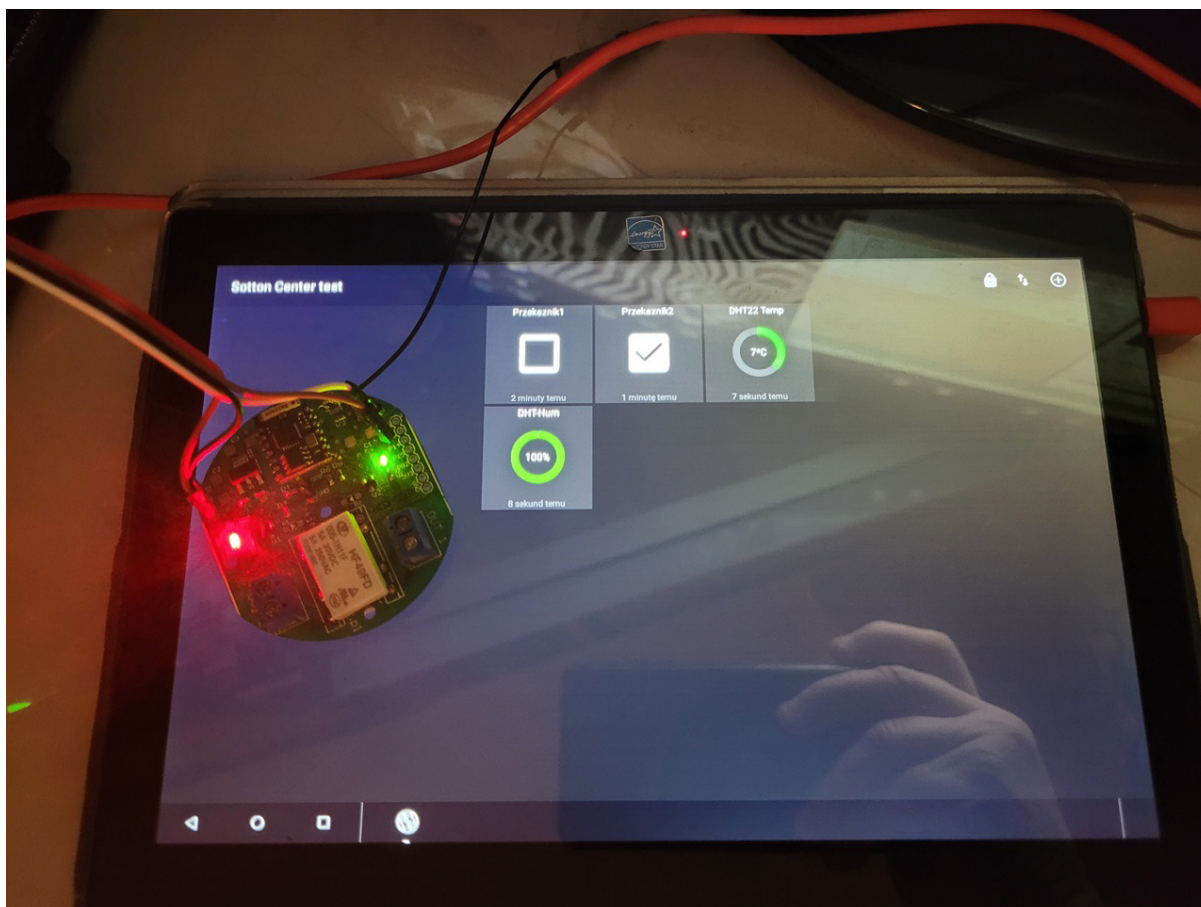
1. Łączy się z siecią WiFi z użyciem hasła i nazwy sieci.
2. Łączy się z naszym brokerem MQTT z użyciem nazwy użytkownika i hasła na porcie 1833.
3. Rozpoczyna subskrybowanie kanałów *wlacznik/przekaznik1* oraz *wlacznik/przekaznik2*.
4. Publikuje dane dotyczące stanów wyjść w kanałach *wlacznik/przekaznik1/status* oraz *wlacznik/przekaznik2/status*.
5. Oczekuje na informacje dostarczane do kanałów subskrybowanych oraz zgodnie z zadaniem zmienia stan odpowiedniego wyjścia i status.

Możemy to obserwować w MQTT Explorerze, który ułatwia nam kontrolowanie tego, co się dzieje na brokerze (rysunek 14.5).



Rysunek 14.5. Podgląd działania sterownika w MQTT Explorerze

A teraz wisienka na torcie. Zobaczcie, jak nasz "system sterowania" wygląda w naszym dashboardzie na smartfonie czy tablecie (rysunek 14.6).



Rysunek 14.6. Podgląd CircleBoard i MQTT Dashboard — włączony kanał 2

Jeśli mamy, tak jak ja, dashboard na smartfonie i na starszym tablecie pełniącym funkcję WallPanela, subskrybowanie statusu jest przydatne, gdyż wtedy widzimy, jaki jest aktualny stan włączników na każdym podłączonym urządzeniu. Na zdjęciu widzicie też nasz DHT22, który u mnie wylądował za oknem i pokazuje parametry na zewnątrz. Niemniej należą się nam gratulacje: mamy działający system *smart home* i dwa prawdziwe urządzenia IoT.

Teraz tylko wyobraźnia ogranicza nas w tworzeniu nowych urządzeń i sposobów sterowania nimi. W tym momencie wystarczy do wyjść przekaźników podłączyć oświetlenie lub

inne urządzenie — oczywiście zgodnie z zasadami bezpieczeństwa — i sterować nim zdalnie.

[3](#) Źródło: Schemat stworzony w narzędziu Fritzing; dostęp 11.06.2024.

[4](#) Źródło: Schemat stworzony w narzędziu Eagle; dostęp 11.06.2024.

[5](#) Źródło: Schemat stworzony w narzędziu Eagle; dostęp 11.06.2024.

15. Ekoelektronika

Tytuł rozdziału zapewne z niczym się Wam nie kojarzy, ale jest jak najbardziej na czasie. Ekoelektronika to wymyślone słowo będące połączeniem elektroniki i ekologii. Czyli jest odpowiedzią na pytanie, jak w świecie elektroniki małymi krokami możemy zmieniać świat. W tym właśnie rozdziale chciałbym Was przekonać do pewnego podejścia, które nawet jeśli nie jest niczym wielkim, może mieć duży wpływ na naszą planetę. Mianowicie chodzi o powtórne użycie niektórych urządzeń, by dać im drugie życie, ale też zmniejszyć ilość elektrośmieci.

Każdy lubi nowoczesne — i nowe — urządzenia, a te starsze często lądują w koszu lub na dnie szuflady. Przy okazji budowania systemu *smart home* warto spojrzeć na niektóre z tych staroci, bo mogą się okazać cennym i tanim elementem systemu. Każdy system *smart home* wymaga jakiegoś WallPanela, za pomocą którego można sterować urządzeniami niezależnie od smartfona, ponadto dobrze mieć szybki podgląd na różne funkcje i je monitorować. Dostępne ekrany HMI czy specjalne panele ścienne z wyświetlaczami są drogie, a my mamy na dnie szuflady starszy model tabletu czy smartfona, który przy odrobinie chęci może być użytecznym urządzeniem naszego systemu. I właśnie teraz chcę Wam przedstawić takie rozwiązanie.

Budujemy WallPanel

U zarania dziejów telefonii GSM jako modemów do instalacji alarmowych (były dostępne specjalne, ale ceny potrafiły rzucić na kolana słonia) używałem... starych nokii i

siemensów. Oczywiście wiadomo, że takie coś musi być cały czas zasilane, a akumulator nie był wysokich lotów, więc zostawianie takiego zestawu działającego na zasilaczu z baterią było niestety pomyłką. Niemniej łatwo można było sobie poradzić z problemem przez podłączenie zasilania do zacisków baterii. Sprzyjała temu konstrukcja wczesnych smartfonów, w których baterię dawało się wyjąć. Ale do sedna powoli dojdziemy.

Budujemy nasz system *smart home*... i potrzebujemy jakiegoś domowego centrum zarządzania. No nie ma sprawy, możemy w domu korzystać ze swojego smartfona — a inni domownicy? Czy nie wygodniej np. mieć taki panel HMI albo coś w tym rodzaju na ścianie lub półce, co pozwoli rzutem oka sprawdzić temperatury, statusy oświetlenia i inne parametry? Tak, to bardzo wygodne i profesjonalne rozwiązanie, a przy okazji eleganckie.

Owszem, możemy kupić wyspecjalizowane urządzenie, ale może w szufladzie leży sobie stary smartfon, któremu padła bateria, ale który poza tym jeszcze jest całkiem spoko? Dodatkowo taki smartfon zapewne będzie z Androidem, a to pozwala całkiem łatwo samemu coś sobie naskrobać lub skorzystać z dostępnych darmowych rozwiązań. Czemu więc go wyrzucać i tym samym stworzyć potencjalnie niebezpieczny dla środowiska odpad, zamiast go użyć?

Oczywiście użycie takiego sprzętu może wydawać się kłopotliwe z powodu zasilania. Na samej ładowarce bez baterii nie będzie działał, zostawienie go w ładowarce z baterią jest mało rozsądne, bo już nie raz doszło do niebezpiecznego wypadku, podłączenie zasilania pod piny baterii też nic nie da, bo sprzęt odpali tylko na chwilę... Czyli nadaje się wyłącznie do kosza, prawda? A co tam, zaśmiecajmy planetę, jednego smartfona więcej nikt nie zauważy. Owszem, jednak gdy to robi sto osób, tysiąc i tak dalej, powstaje naprawdę poważny problem w postaci masy

elektrośmieci, już nie wspominając o pośrednich zagrożeniach, np. pożarami. Ale jest nadzieja. Wiem, wiem, wielu z Was pomyśli w tym momencie: "Nadzieja matką głupich". I słusznie, często jednak wystarczy się przyjrzeć problemowi.

Ja w swoich szpargałach miałem akurat jakiegoś huaweia P8 Lite. Bateria spuchła nieco i zero trzymania, więc postanowiłem pokombinować i tym sposobem metodą prób i błędów doszedłem do prostego rozwiązania. Potrzebujemy:

- starego smartfona lub tabletu,
- nożyka,
- lutownicy,
- potrzeby stworzenia takiego ustrojstwa lub — jak kto woli — 10 kg chęci do pracy (ostatnio widziałem w Biedronce w promocyjnych cenach w 25-kilogramowych workach).

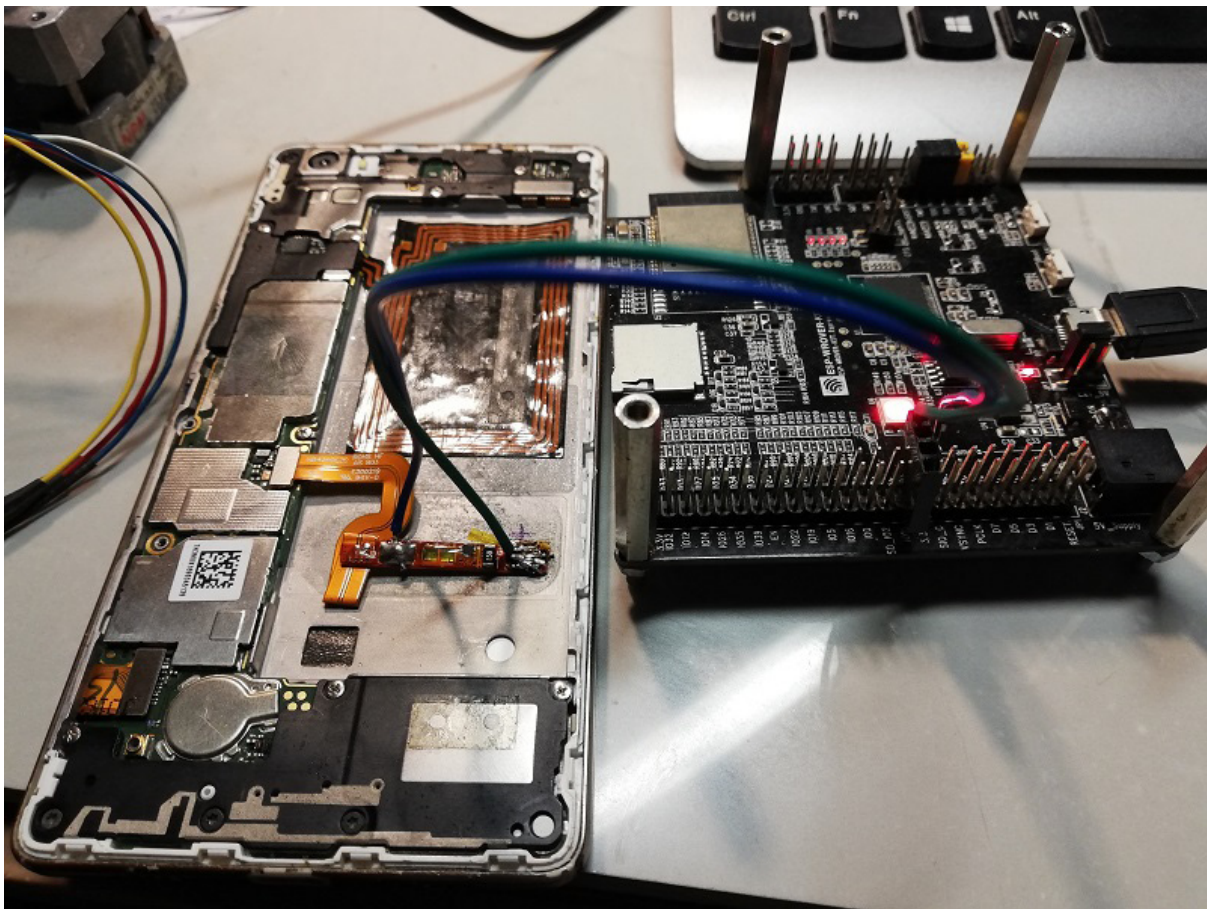
Gdy już mamy wszystko, czego nam potrzeba, musimy się dostać do środka. Łatwo wyszukać w sieci informacje o tym, jak otworzyć dane urządzenie, więc nie będę tu niepotrzebnie się rozpisywał. Musimy jednak postępować ostrożnie, zwłaszcza z baterią, by jej nie przedziurawić. Gdy już uda się zdjąć pokrywę, ostrożnie odpinamy i wyjmujemy baterię. Teraz musimy dostać się do złączy samego pakietu LiPo, więc delikatnie odwijamy zabezpieczenia od strony złącza. Gdy już mamy dostęp do kontrolera baterii, miernikiem sprawdzamy, gdzie jest plus, i zaznaczamy. Teraz musimy delikatnie odciąć blaszki pakietu od elektroniki, tak by nie uszkodzić torebki baterii.

Uf, udało się... Możemy bezpiecznie utylizować baterię (w przeznaczonym do tego miejscu).

Sam kontroler natomiast jest nam potrzebny, dlatego że ma wtyki pasujące do płyty i dodatkowo pozwoli nam oszukać

system Androida i jego obwody zasilania.

Nasza typowa bateria ma napięcie 3,7 V. W P8 na zaciskach za kontrolerem jest 3,95 V, czyli do płyty głównej dochodzi 3,72 V, a podczas ładowania baterii na zaciskach jest napięcie 4,89 V. Zatem lutujemy przewody do punktów, gdzie był podłączony pakiet LiPo, i wpinamy tak spreparowany układ do płyty w smartfonie (rysunek 15.1).

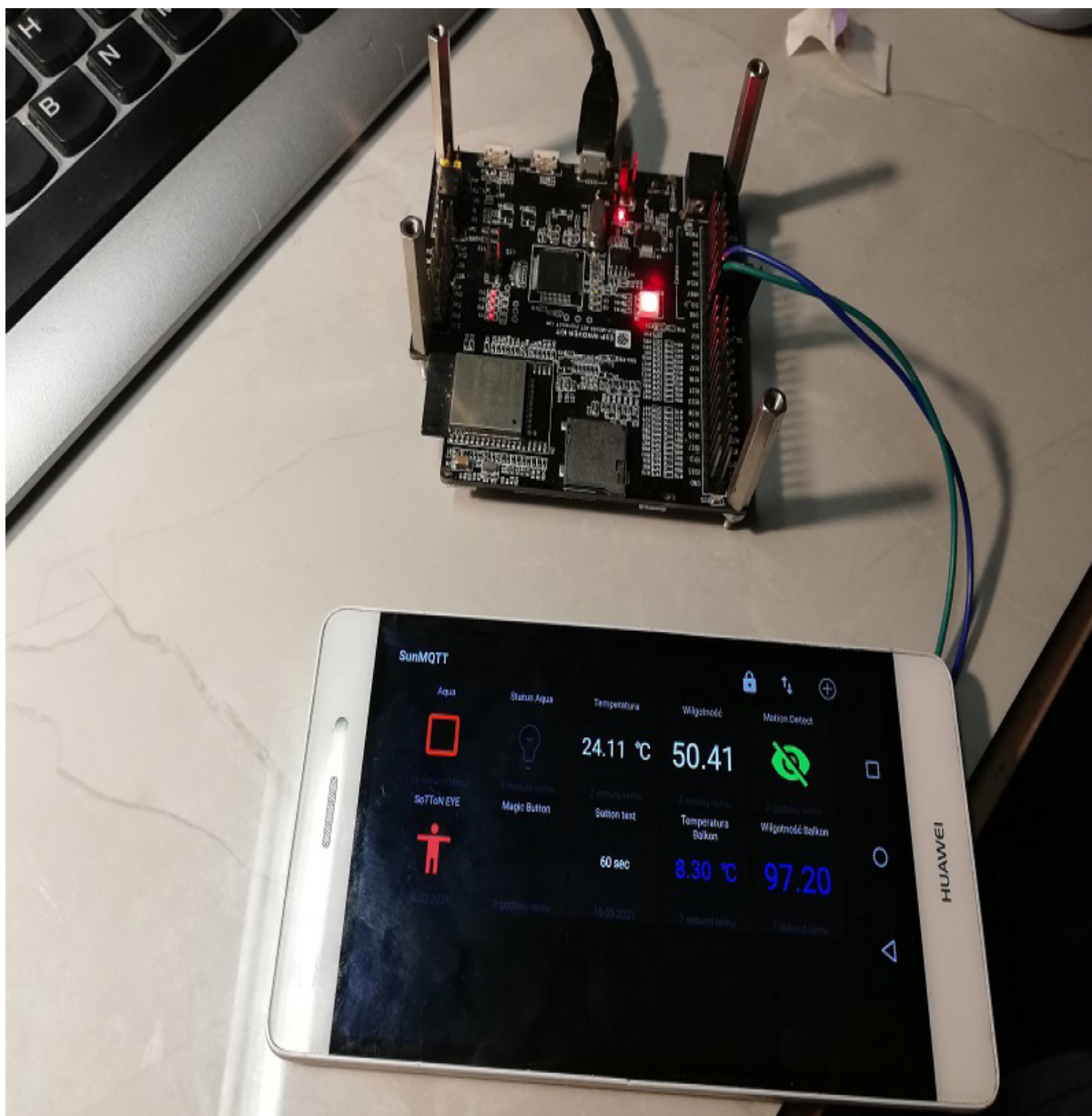


Rysunek 15.1. Smartfon przygotowany do pracy ciągłej

Tak na razie wygląda to u mnie. Ze względu na poziom wspomnianych napięć postanowiłem zaryzykować i podłączyłem się pod zaciski 5 V na wrover-kicie zasilanym z USB.

Prąd 500 mA, a nawet mniejszy, bo trochę połknie płytka z ESP32, nie powinien ubić naszego przyszłego panelu sterującego. Ostatecznie i tak jest tam 4,98 V, więc różnica względem napięcia ładowania jest niewielka.

Naciskamy przycisk zasilania. System powinien się uruchomić (rysunek 15.2).

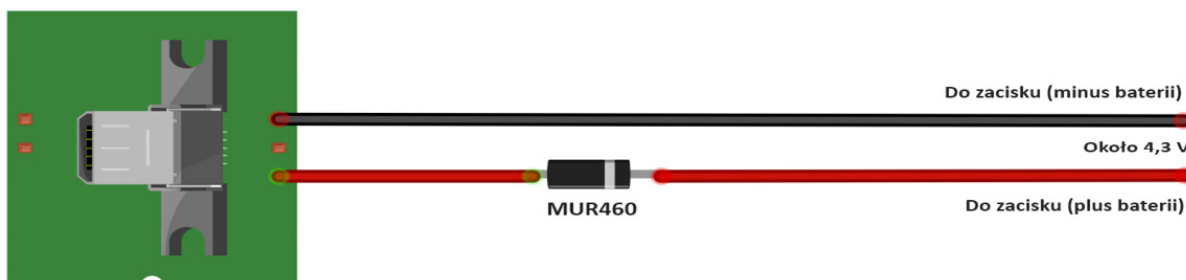


Rysunek 15.2. System uruchomiony

Operacja się udała i nasz pacjent żyje. Uruchomiony został MQTT Dashboard. Tym sposobem mamy kolorowy LCD o przyzwoitej rozdzielczości, dotyk, WiFi, BT, RAM-u nie brakuje i jeszcze GSM... do wykorzystania choćby jako bramka powiadomień SMS-em w sytuacjach alarmowych. Całość działa poprawnie. Jednak w niektórych przypadkach, mimo że system pokazuje pełne naładowanie baterii, 100%, jego poziom spada. Wynika to z budowy kontrolera i systemu. Tu z pomocą przychodzi mała sztuczka.

Na zasilacz wybrałem ładowarkę dającą 5 V, 2 A, która jest podłączona do USB w telefonie, a linię +5 V z gniazda USB podłączyłem przewodem poprzez diodę MUR460 do dodatniego zacisku baterii. Powoduje to, że tak zmodyfikowany telefon cały czas utrzymuje poziom baterii na 85%.

Działając na stałym zasilaniu, można bezpiecznie zamontować takie urządzenie w jakiejś zgrabnej obudowie — i mamy ładny, dobrze działający panel kontrolny. Schemat połączenia jest pokazany na rysunku 15.3.



Rysunek 15.3. Schemat połączeń⁶

Rzecz jasna jeszcze prościej jest ze starszymi tabletami, takimi jak Trekstor, Manta, Pentagonam. Ale jak widzicie,

można uzyskać coś fajnego także ze starszego, już niepotrzebnego sprzętu. Warto się zainteresować tematem pozyskiwania. I jest taniej dla nas, i lepiej dla natury. Oczywiście, jeśli nie czujecie się na siłach wykonać takich modyfikacji w tablecie lub telefonie, jest inna opcja, która jest znacznie prostsza do zastosowania, a daje jeszcze inne możliwości.

WallPanel — wariant drugi

Jak już wspomniałem, opisana metoda może być zbyt skomplikowana w przypadku niektórych urządzeń, zwłaszcza co bardziej renomowanych producentów. Ale jest jeszcze jeden, łatwy sposób na wykonanie panelu kontrolnego. Mianowicie możemy wykorzystać radio samochodowe z Androidem. Powiecie: ooo, panie, to koszty! Niekoniecznie, moi drodzy. Nieco tandetne używane radyjko często można znaleźć w okolicy 100 złotych w znanych witrynach aukcyjnych czy ogłoszeniowych. Do samochodu średnio się takie tanie chińskie urządzenia nadają, ale jako panel — owszem. Zadanie ułatwia nam brak baterii i zasilanie 12 V. Tanie tzw. radia 2 DIN mają nieduże rozmiary i często 7-calowy ekran. Dodatkowo możemy korzystać z radia, dołączyć kamerkę i wiele innych. Takie rozwiązanie też nada się do naszych celów.

[6](#) Źródło: Schemat stworzony w narzędziu Fritzing; dostęp 11.06.2024.

16. Dashboard

Skoro mamy panel, musimy się teraz postarać o oprogramowanie. Na nasze szczęście dla Androida jest tego mnóstwo. Ja polubiłem MQTT Dash firmy Routix Software, dostępny w sklepie Google Play. Jest lekki, nie zawiera reklam i ma spore możliwości konfiguracyjne. Ale bez problemu znajdziecie coś innego, co będzie dla Was odpowiednie, jak również możecie napisać własny program, jeśli się czujecie na siłach. Z ciekawych dostępnych opcji wymienimy choćby:

- My MQTT,
- MQTT Panel IoT,
- MQTT Client,
- IoT MQTT Panel,
- MQTT Alert.

Jest też wiele innych, a każdy ma swoje wady i zalety. Warto zatem spróbować różnych rozwiązań, by móc samodzielnie ocenić, która aplikacja będzie najlepsza.

Wróćmy do naszego przykładowego sterownika i czujnika. Skonfigurujemy panel, by wszystko działało poprawnie. Całość przedstawię na przykładzie MQTT Dash.

Po pobraniu i zainstalowaniu uruchamiamy aplikację. Niewiele się tu dzieje, ale klikamy ikonkę plusa. W ten sposób dodamy połączenie z naszym brokerem. Pojawi się karta, na której musimy wypełnić kilka pól:

- *Name* — tu wpisujemy "przyjazną" nazwę naszego serwera. Może być właściwie dowolna.
- *Address* — tu podajemy adres IP naszego brokera. W przypadku gdy sprzęt działa lokalnie, będzie to lokalny

adres IP. W smartfonie warto ustawić, po odpowiednim skonfigurowaniu routera, adres zewnętrzny — to nam umożliwi dostęp do systemu z dowolnego miejsca.

- *Port* — domyślnie jest to 1883, ale w razie potrzeby możemy go zmienić na inny, np. 3000.
- Jeśli używamy — możemy włączyć SSL/TLS.
- *User name* — tu podajemy nazwę użytkownika MQTT.
- *User password* — tu podajemy hasło do brokera.
- *Client ID* — możemy zostawić domyślne ustawienie. To nazwa klienta MQTT, np. *Panel Kuchnia*.

Pozostałych opcji na razie nie musimy zmieniać. Klikamy zatem ikonkę dyskietki, a wtedy nasze dane się zapiszą. A nasz serwer pojawi się na liście pod wybraną nazwą. Zarządzać możemy wieloma połączeniami z różnymi brokerami, które dodamy do listy.

Teraz wybieramy nasze połączenie i otrzymujemy pusty ekran, który wypełnimy kontrolkami, przyciskami i co tam będzie nam potrzebne. Tradycyjnie u góry klikamy ikonkę plusa. Pojawi się menu. Tu wybieramy, co jest potrzebne do danego zadania. Do dyspozycji mamy:

- *Text*,
- *Switch/button*,
- *Range/progress*,
- *Multi-choice*,
- *Image*,
- *Color*.

Do sterowania przełącznikami, jak łatwo się domyślić, będziemy potrzebować przycisku. Wybieramy więc *Switch/button* i ustawiamy:

1. *Name* — nazwa przycisku, np. *Lampa w altanie*.
2. *Topic (sub)* — temat, który będziemy subskrybować, czyli nasz temat ze statusem. Wpisujemy więc w polu **włącznik/przekaznik1/status**.

3. Stawiamy ptaszek przy *Enable publishing*.
4. *Topic (pub)* — wpisujemy kanał, w którym będziemy publikować informacje, czyli **włącznik/przekaznik**.
5. Zaznaczamy pole wyboru przy *update metric on publish* i wartości *On* i *Off* ustawiamy, odpowiednio, na **1** i **0**.
6. Możemy też zmienić przez kliknięcie pól *On* i *Off* ikony włącznika oraz kolory, np. kliknięciem białego kwadratu pod *On* otworzymy okno *Select a color*, gdzie wybierzemy kolor zielony, analogicznie dla *Off* — czerwony.
7. Klikamy ikonę dyskietki w celu zapisania naszych ustawień.

Nasz pierwszy przycisk się pojawił. Możemy teraz sprawdzić, czy działa. Bardzo ładnie zmienia kolory, a nasz przekaźnik klika, czyli działa. Drugi przycisk dodamy nieco prościej: przytrzymując palec na już dodanym przycisku, wybieramy z menu *Clone*, i oto mamy drugi przycisk. Teraz to jego przytrzymujemy i wybieramy *Edit*. Wystarczą niewielkie zmiany, takie jak zmiana nazwy i cyfry 1 w tematach na 2. Następnie klikamy ikonę dyskietki, i gotowe. Nasze przełączniki działają. Jeśli teraz zrobimy to samo na innym urządzeniu, to zobaczymy, że gdy włączymy na jednym z nich urządzenie, np. *Lampa w altanie*, drugie urządzenie od razu będzie wiedziało, że ta lampa jest włączona, i kolor włącznika zmieni się na zielony. To dlatego, że subskrybujemy temat *status*. W przeciwnym razie, jeśli będziemy chcieli zgasić światło na drugim urządzeniu, będziemy musieli kliknąć dwa razy, czyli włączyć i dopiero wtedy wyłączyć światło, urządzenie bowiem nie będzie wiedziało, że światło już jest włączone. Teraz możecie się pobawić całością i powybierać ikonki, by się przyzwyczaić do konfiguracji. Jak widzicie, możemy prosto i łatwo sterować naszym IoT i *smart home*.

17. Sterowanie ogrzewaniem, na przykład w akwarium

Pobawiliśmy się prostymi przykładami, teraz zbudujemy prosty sterownik grzałek, np. akwariowych. Nasz sterownik będzie włączał grzałki w zależności od temperatury, którą ustawimy w naszym WallPanelu. Tym razem użyjemy płytki z ESP32, np. ESP32 One. Program napiszemy z użyciem frameworka IDF i systemu FreeRTOS.

Nasz kod oczywiście będzie prosty i zapewne będzie wymagał dostosowania do konkretnego zadania. Użyjemy czujnika temperatury DS18B20 w wodoszczelnej obudowie, który łatwo znaleźć na znanym portalu aukcyjnym. DS18B20 korzysta z magistrali 1-Wire i jest dokładnym cyfrowym termometrem. Do sterowania temperaturą w akwarium jego parametry są aż za dobre. Oto uproszczone dane techniczne czujnika DS18B20:

- napięcie zasilania: od 3,0 V do 5,5 V,
- zakres pomiarowy: od -55°C do 125°C ,
- dokładność: $\pm 0,5^{\circ}\text{C}$ w zakresie od -10°C do 85°C ,
- rozdzielczość: od 9 do 12 bitów.

Grzałki będą włączane przekaźnikiem, takim jaki już znacie, a ponieważ moc nie jest tu duża, wystarczy nam przekaźnik 10 A. To tyle od strony sprzętowej; teraz musimy trochę poczarować w kodzie. Kod utworzymy w Visual Studio Code z użyciem zainstalowanych wcześniej narzędzi Espressif. Do obsługi DS18B20 będziemy musieli zainstalować — jeśli nie chcemy pisać własnej biblioteki — oficjalny sterownik. Nie jest to skomplikowana operacja, niemniej pokrótce ją opiszę.

Instalacja sterownika DS18B20

Mamy już utworzony projekt. Teraz musimy zainstalować dodatkowe oprogramowanie, które umożliwi nam korzystanie z termometru DS18B20 w naszym kodzie. Oczywiście, jak wspomniałem, możemy bibliotekę napisać samodzielnie, ale skoro jest gotowa oficjalna wersja, użyjemy jej.

Wersja pierwsza

1. Przechodzimy do folderu naszego projektu (ścieżka jest podawana przy tworzeniu projektu).
2. Tworzymy w głównym katalogu projektu folder *components*.
3. W nim tworzymy folder *ds18x20* i wewnątrz umieszczamy pliki *ds18x20.c* oraz *ds18x20.h* z repozytorium ESP-IDF dostępnego na oficjalnym GitHubie pod adresem <https://github.com/UncleRus/esp-idf-lib/tree/master/components/ds18x20>.
4. Teraz musimy włączyć sterownik w pliku *CMakeLists.txt*, który znajduje się w folderze głównym naszego projektu. Plik ten powinien zawierać następującą treść:

```
# For more information about build system see
# https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/build-system.html
# The following five lines of boilerplate have to be in
# your project's
# CMakeLists in this exact order for cmake to work
correctly
cmake_minimum_required(VERSION 3.5)
set(EXTRA_COMPONENT_DIRS components/ds18x20)
```

```
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(ESP32-kontroler-temperatury-w-akwarium)
set(COMPONENT_ADD_INCLUDEDIRS
components/ds18b20/include)
target_compile_features(${CMAKE_PROJECT_NAME} PUBLIC
cxx_std_17)
```

Od tej pory możemy używać biblioteki DS18x20 po umieszczeniu w kodzie źródłowym `#include "ds18x20.h"`. Oczywiście jest nam jeszcze potrzebny sterownik magistrali 1-Wire, którego używa właśnie dodana biblioteka, więc idąc na skróty, możemy pobrać pliki *onewire.c* i *onewire.h* oraz dodać je do naszego folderu *ds18x20*, i to wszystko. Możemy też utworzyć folder *onewire* i umieścić w nim pliki sterownika magistrali oraz dodać do pliku *CMakeList.txt* dwie linijki o treści:

```
set(EXTRA_COMPONENT_DIRS components/onewire)
set(COMPONENT_ADD_INCLUDEDIRS
components/onewire/include)
```

Wersja druga

Tu wykorzystujemy klonowanie repozytoriów z użyciem konsoli CMD i aplikacji Git. Nie będę się tu specjalnie rozpisywał, gdyż wszystko znajdziecie na stronie <https://esp-idf-lib.readthedocs.io/en/latest/index.html>.

Warto się z nią zapoznać, zapewnia bowiem sporą dawkę wiedzy, która może być nam potrzebna.

Wykonajmy zatem poniższe kroki:

1. Otwieramy konsolę CMD i przechodzimy do folderu naszego projektu:

```
cd cd ~/uzytkownik/work/nasz_projekt
```

2. Klonujemy repozytorium przez wpisanie:

```
git clone https://github.com/esp8266/esp-idf-lib.git components/esp-idf-lib
```

3. Po chwili mamy sklonowane wszystkie komponenty, więc modyfikujemy plik *CMakeList.txt*:

```
make_minimum_required(VERSION 3.5)
set(EXTRA_COMPONENT_DIRS components)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(your_project_name)
set(COMPONENT_REQUIRES esp-idf-lib)
set(COMPONENT_SRCDIRS "main")
set(COMPONENT_ADD_INCLUDEDIRS ".")
register_component()
```

Mamy już dostęp do wszystkich sterowników zawartych w repozytorium w naszym projekcie. Ja używam właśnie tej wersji.

Uf, nie była to skomplikowana operacja. Teraz możemy przejść dalej i pisać nasz magiczny kod, który tchnie życie w układ ESP32. I on będzie odtąd służyć naszym. Żarty żartami, ale czas na poważne pisanie. Trochę tego będzie.

Kod kontrolera

W naszym kodzie użyjemy znanych Wam już metod inicjowania połączenia z siecią WiFi i logowania do naszego brokera MQTT, na którym będziemy mogli, przez publikowanie w odpowiednich tematach, ustawiać temperaturę lub sterować ręcznie grzałkami. Zatem do dzieła. Dużo pisania nie ma. Nasz kod wygląda następująco:

```
#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
```

```

#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "freertos/semphr.h"
#include "freertos/queue.h"
#include "freertos/xtensa_api.h"
#include "driver/gpio.h"
#include "mqtt_client.h"
#include "ds18b20.h"

#define WIFI_SSID  "<nazwa sieci WiFi>"
#define WIFI_PASSWORD "<hasło sieci WiFi>"
#define MQTT_BROKER  "<adres MQTT>"
#define MQTT_PORT  1883
#define MQTT_USER  "<użytkownik MQTT>"
#define MQTT_PASSWORD "<hasło MQTT>"
#define MQTT_TOPIC  "kontroler/"
#define GPIO_GRZALKA_1 4
#define TEMP_MINIMUM 25.0
#define DS18B20_PIN GPIO_NUM_12

static esp_mqtt_client_handle_t mqtt_client;
static float setpoint_temp = TEMP_MINIMUM;
static int manual_control = 0;
static void initialize_wifi(void)
{
    tcpip_adapter_init();
    esp_event_loop_create_default();
    wifi_init_config_t wifi_init_config =
WIFI_INIT_CONFIG_DEFAULT();
    esp_wifi_init(&wifi_init_config);

```

```

    esp_wifi_set_storage(WIFI_STORAGE_RAM);
    wifi_config_t wifi_config = {
        .sta = {
            .ssid = WIFI_SSID,
            .password = WIFI_PASSWORD
        },
    };
    esp_wifi_set_mode(WIFI_MODE_STA);
    esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config);
    esp_wifi_start();
}

static esp_err_t
mqtt_event_handler(esp_mqtt_event_handle_t event)
{
    switch (event->event_id) {
        case MQTT_EVENT_CONNECTED:
            ESP_LOGI("MQTT", "Połączono z brokerem MQTT");
            break;
        case MQTT_EVENT_DISCONNECTED:
            ESP_LOGI("MQTT", "Rozłączono z brokerem MQTT");
            break;
        case MQTT_EVENT_DATA:
            handle_mqtt_data(event->data, event->topic,
event->data_len);
            break;
        default:
            break;
    }
}

```

```

    return ESP_OK;
}

static void handle_mqtt_data(const char *data, const
char *topic, int data_len)
{
    if (strncmp(topic, MQTT_TOPIC "/ustaw_temp",
strlen(MQTT_TOPIC "/ustaw_temp")) == 0)
    {
        // Ustaw temperaturę
        setpoint_temp = atof(data);
        ESP_LOGI("MQTT", "Ustawiono temperaturę na
%.2f", setpoint_temp);
    }
    else if (strncmp(topic, MQTT_TOPIC
"/manual_control", strlen(MQTT_TOPIC
"/manual_control")) == 0)
    {
        // Ustaw tryb ręcznego sterowania
        manual_control = atoi(data);
        ESP_LOGI("MQTT", "Tryb ręcznego sterowania:
%d", manual_control);
        // W zależności od trybu ręcznego sterowania,
ustaw grzałki
        if (manual_control == 1)
        {
            gpio_set_level(GPIO_GRZALKA_1, 1);

        }
        else {
            gpio_set_level(GPIO_GRZALKA_1, 0);
        }
    }
}

```

```

        }
    }
}

static void initialize_mqtt(void)
{
    esp_mqtt_client_config_t mqtt_cfg =
    {
        .uri = "mqtt://" MQTT_BROKER ":" MQTT_PORT,
        .username = MQTT_USER,
        .password = MQTT_PASSWORD,
    };
    mqtt_client = esp_mqtt_client_init(&mqtt_cfg);
    esp_mqtt_client_register_event(mqtt_client,
    ESP_EVENT_ANY_ID, mqtt_event_handler, mqtt_client);
    esp_mqtt_client_start(mqtt_client);
}

float read_ds18b20()
{
    float temp = 0.0;
    ds18b20_init(DS18B20_PIN);
    if (ds18b20_measure_and_read(DS18B20_PIN, &temp))
    {
        ESP_LOGI("TEMP", "Temperatura: %.2f", temp);
        return temp;
    }
    else {

```

```

        ESP_LOGE("TEMP", "Błąd odczytu temperatury z
DS18B20");
        return -1.0; // Wartość błędu
    }
}

void app_main(void)
{
    initialize_wifi();
    initialize_mqtt();
    while (1)
    {
        float current_temp = read_ds18b20();
        if (current_temp >= 0.0) {
            if (manual_control == 0) {
                // Automatyczne sterowanie grzałkami w
zależności od temperatury
                if (current_temp < setpoint_temp)
                {
                    gpio_set_level(GPIO_GRZALKA_1, 1);

                }
            }
            else {
                gpio_set_level(GPIO_GRZALKA_1, 0);
            }
        }
        // Publikuj bieżącą temperaturę na MQTT
        char temp_str[10];
        snprintf(temp_str, sizeof(temp_str),
        "%.2f", current_temp);
    }
}

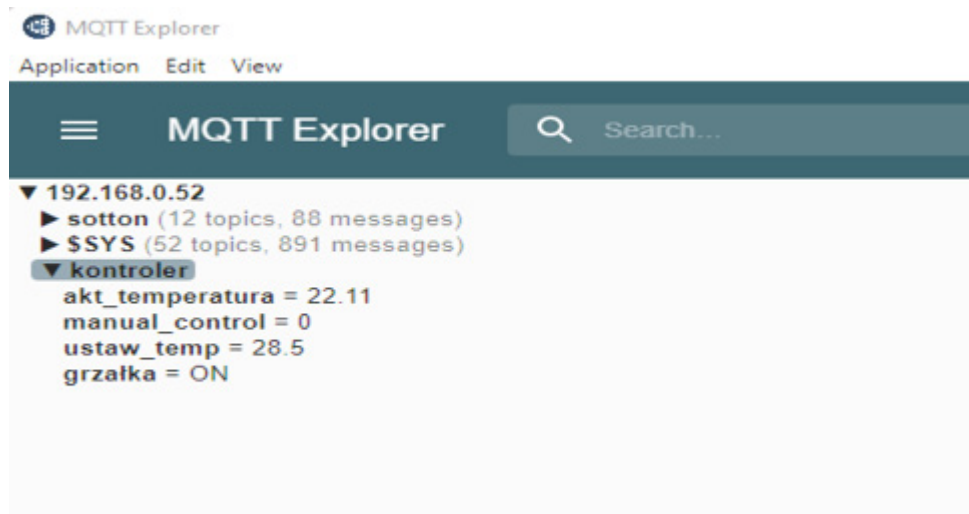
```

```

        esp_mqtt_client_publish(mqtt_client,
MQTT_TOPIC "/akt_temperatura", temp_str, 0, 1, 1);
    }
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}
}

```

Jak widzicie, kod nie jest specjalnie skomplikowany, ale umożliwia nam monitorowanie temperatury wody w akwarium w temacie *kontroler/akt_temperatura*. Dodatkowo możemy przełączać grzałkę w tryb sterowania ręcznego *ON/OFF* i ustawiać temperaturę przez publikowanie w temacie *kontroler/ustaw_temp*. Na rysunku 17.1 jest pokazane, jak to wygląda w MQTT Explorerze.



Rysunek 17.1. Podgląd komunikatów kontrolera

Omówienie kodu

Aby tradycji stało się zadość, krótko w punktach omówimy funkcje naszego kodu, co ułatwi też zrozumienie jego

działania początkującym. Większość kodu już znacie, więc zrobimy tak:

1. `initialize_wifi` — inicjalizuje połączenie z siecią WiFi.
2. `mqtt_event_handler` — obsługuje zdarzenia związane z połączeniem MQTT, takie jak połączenie i rozłączenie.
3. `handle_mqtt_data` — obsługuje dane otrzymane z MQTT, w tym ustawienia temperatury i trybu ręcznego sterowania.
4. `initialize_mqtt` — inicjalizuje klienta MQTT.
5. `read_temperature` — odczytuje temperaturę z czujnika DS18B20.
6. `app_main` — funkcja główna programu, w której zawarta jest główna pętla, obsługująca logikę sterowania grzałkami w zależności od temperatury i danych otrzymanych z MQTT.

Oczywiście to bardzo prosta wariacja, niemniej po dostosowaniu i dodaniu kilku parametrów można zbudować kontroler akwariów z prawdziwego zdarzenia. Dodać można sterowanie oświetleniem dzień/noc, kontrolę natleniania i wiele innych parametrów. I jeśli chcieliście coś takiego zrobić, a nie wiedzieliście jak, już macie gotową bazę do rozbudowy. U mnie np. rozwinięciem sterownika akwariów do hodowli rybek jest sterownik terrarium. Jak widzicie, możliwości mimo niewielkiego rozmiaru kodu są ogromne.

18. Monitorowanie zużycia energii

W dzisiejszych czasach monitorowanie zużycia energii elektrycznej jest dość istotne, nie tylko z powodów klimatycznych, ale też ekonomicznych. Nasze urządzenie będzie więc monitorować określone inne urządzenie lub pomieszczenie i publikować dane za pomocą MQTT, co pozwoli nam na bieżącą obserwację i minimalizowanie zużycia energii oraz redukcję kosztów.

Dla bezpieczeństwa pomiary zrealizujemy z zastosowaniem popularnego czujnika natężenia dla prądu przemiennego AC SCT 013-030, który umożliwia bezinwazyjny pomiar w zakresie do 30 A. Czujnik ma wbudowany rezystor bocznikowy, dzięki czemu na wyjściu pojawia się napięcie o amplitudzie proporcjonalnej do mierzonego natężenia w skali 0 do 30 A, czyli w zakresie 0 do 1 V. Przekładnik umożliwia zamontowanie bez rozcinania przewodu, na którym będziemy dokonywać pomiaru, o średnicy do 13 mm. Dzielony rdzeń magnetowodu wraz z klipsem jest idealny do zapięcia na przewodzie, a obwód przekładnika jest odizolowany od otoczenia barierą galwaniczną o wytrzymałości 1500 VAC.

Oto krótka charakterystyka czujnika AC SCT 013-030:

- prąd wejściowy: od 0 do 30 A AC,
- napięcie wyjściowe: od 0 do 1 V,
- wbudowany rezystor obciążający: 62 Ω ,
- stosunek uzwojenia: 1800 : 1,
- zakres temperatury pracy: od -25°C do 70°C ,
- długość przewodu: 1 m,
- masa czujnika: ok. 45 g,

Resztę parametrów łatwo znajdziecie w sieci, tymczasem zajmijmy się kodem.

Kod programu

Kod napiszemy w ESP-IDF dla ESP32. Umożliwi nam monitorowanie prądu przemiennego i będzie publikował wyniki pomiarów na brokerze MQTT, co umożliwi nam łatwe śledzenie poprzez dashboard. Oczywiście program można rozbudować i w odpowiedni sposób dostosować do własnych potrzeb. Całość prezentuje się następująco:

```
#include <stdio.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <freertos/event_groups.h>
#include <freertos/semphr.h>
#include <driver/adc.h>
#include <driver/gpio.h>
#include <esp_log.h>
#include <mqtt_client.h>
#define TAG "ACCurrentMonitor"
#define MQTT_BROKER "<adres brokera MQTT>"
#define MQTT_PORT 1883
#define MQTT_USER "<użytkownik MQTT>"
#define MQTT_PASSWORD "<hasło MQTT>"
#define MQTT_TOPIC "zuzycie_energii"
#define ADC_PIN GPIO_NUM_34 // Ustawienie pinu ADC
static esp_mqtt_client_handle_t mqtt_client;
void mqtt_app_start(void)
```

```

{
    esp_mqtt_client_config_t mqtt_cfg = {
        .uri = MQTT_BROKER,
        .username = MQTT_USER,
        .password = MQTT_PASSWORD,
    };
    mqtt_client = esp_mqtt_client_init(&mqtt_cfg);
    esp_mqtt_client_start(mqtt_client);
}

void setup_adc()
{
    adc1_config_width(ADC_WIDTH_BIT_12);
    adc1_config_channel_atten(ADC_CHANNEL_6,
ADC_ATTEN_DB_11);
}

void publish_current(float current)
{
    char data[50];
    snprintf(data, sizeof(data), "{\"prad\": %.2f}",
current);
    esp_mqtt_client_publish(mqtt_client, MQTT_TOPIC,
data, 0, 1, 0);
}

void read_sensor_data(void *pvParameters)
{
    while (1) {
        uint32_t adc_value =
adc1_get_raw(ADC_CHANNEL_6);

```

```

        float current = (adc_value / 4096.0) * 3.3; //
        Konwersja wartości ADC
        // na napięcie

        ESP_LOGI(TAG, "Prąd przemienny: %.2f A",
current);

        publish_current(current);
        vTaskDelay(10000 / portTICK_PERIOD_MS);
    }
}

void app_main()
{

    ESP_ERROR_CHECK(adc1_config_width(ADC_WIDTH_BIT_12));

    ESP_ERROR_CHECK(adc1_config_channel_atten(ADC_CHANNEL_6
, ADC_ATTEN_DB_11));

    ESP_ERROR_CHECK(esp_log_level_set("*",
ESP_LOG_INFO));

    wifi_config_t wifi_config = {
        .sta = {
            .ssid = "<sieć WiFi>",
            .password = "<hasło sieci>",
        },
    };

    ESP_ERROR_CHECK(esp_wifi_init(&wifi_config));
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_start());
    mqtt_app_start();

    xTaskCreate(&read_sensor_data, "read_sensor_data",
4096, NULL, 5, NULL);

```

```
}
```

Kod może nie jest imponujący, ale wywiązuje się ze swojego zadania.

Omówienie kodu

Program jest tak prosty, że zasadniczo nie mamy co omawiać, ale dla porządku omówimy funkcję odczytu czujnika, tak by wyjaśnić, co się tam dzieje. Funkcja `read_sensor_data` jest zaimplementowana jako zadanie (task) w systemie FreeRTOS. Jej działanie jest stosunkowo proste, ale dla potomnych objaśnimy je.

Nasza funkcja zawiera nieskończoną pętlę `while(1)`, co oznacza, że będzie się wykonywać w kółko, dopóki zadanie będzie aktywne, czyli będzie stale zbierać dane z czujnika.

Odczyt wartości z ADC (ang. *Analog-to-Digital Converter*, konwerter analogowo-cyfrowy) następuje w tym oto wierszu:

```
uint32_t adc_value = adc1_get_raw(ADC_CHANNEL_6);
```

Odczytujemy ją z kanału `ADC_CHANNEL_6`. Tym samym `adc1_get_raw` zwraca bezpośrednio z konwertera wartość, która jest liczbą całkowitą z zakresu od 0 do 4095. Wartość tę musimy skonwertować na napięcie:

```
float current = (adc_value / 4096.0) * 3.3;
```

Właśnie w tym wierszu wartość z ADC jest przeliczana na napięcie. Zakładamy, że ADC ma rozdzielczość 12 bitów (4096 możliwych wartości) i działa w zakresie 0 – 3,3 V. Wartość odczytaną (`adc_value`) dzielimy przez 4096,0, a następnie mnożymy przez 3,3. W wyniku otrzymamy napięcie wyrażone w woltach.

```
ESP_LOGI(TAG, "Prąd przemienny: %.2f A", current);
```

Wartość przeliczonego prądu jest logowana z użyciem funkcji `ESP_LOGI` do systemu wraz z poziomem informacji (INFO). `%f` oznacza format do wypisywania liczb zmiennoprzecinkowych, a zapis `%.2f` ogranicza liczbę miejsc po przecinku do dwóch.

```
publish_current(current);
```

Funkcja `publish_current` jest wywoływana w celu opublikowania przeliczonej wartości prądu wyrażonej w amperach w naszym temacie na brokerze MQTT.

```
vTaskDelay(10000 / portTICK_PERIOD_MS);
```

Po wykonaniu odczytu i opublikowaniu danych zadanie jest uśpione (ang. *delayed*) na mniej więcej 10 sekund, zanim rozpocznie kolejny cykl. `vTaskDelay` jest funkcją z FreeRTOS, która opóźnia wykonanie zadania na określony czas, co jest konieczne, by wyniki nie były zafałszowane i nie wystąpiły błędy konwersji.

W rezultacie funkcja `read_sensor_data` odczytuje wartość z kanału ADC, przelicza ją na napięcie, loguje i publikuje na brokerze, a następnie czeka przed ponownym rozpoczęciem cyklu pomiarowego.

Oczywiście same ampery mogą nam nie wystarczyć, dlatego warto nieco zmodyfikować funkcję, by poza amperami publikowała też dane wyrażone w watogodzinach. W tym celu wystarczy skorzystać ze znanego już Wam wzoru $P = I \cdot V$. Modyfikowania kodu nie ma za dużo. Wystarczy, że na samym początku programu, w sekcji definicji, dodamy definicję napięcia sieci:

```
#define VOLTAGE 230.0
```

Następnie dodamy funkcję publikowania mocy, której nie muszę omawiać, bo już wiecie, co wykonuje:

```
void publish_power(float power)
{
```

```

    char data[50];
    snprintf(data, sizeof(data), "{\"moc\": %.2f}",
power);
    esp_mqtt_client_publish(mqtt_client, MQTT_TOPIC,
data, 0, 1, 0);
}

```

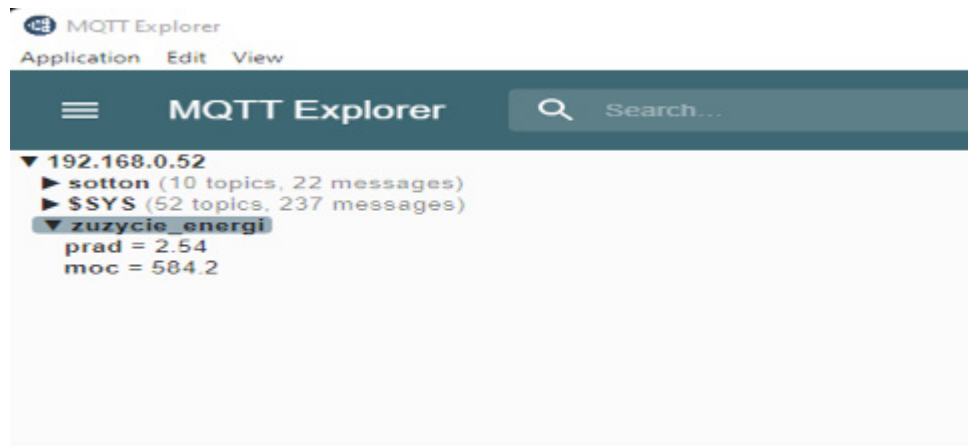
W funkcji `read_sensor_data`, służącej do odczytu czujnika, wystarczy dodać przeliczenie prądu na moc:

```

// Przeliczanie prądu na moc
    float power = current * VOLTAGE;
    ESP_LOGI(TAG, "Moc: %.2f W", power);
    publish_power(power);

```

I gotowe. W naszym brokerze zobaczymy dane dotyczące pobieranego prądu i mocy (rysunek 18.1).



Rysunek 18.1. Podgląd komunikatów monitora

Prawda, że proste? Oczywiście, to prosty pomiar, a wynik zależy od kalibracji czujnika i fizycznego napięcia w sieci. (Można by jeszcze dodać pomiar napięcia, by mieć jak najbardziej szczegółowe dane). Niemniej właśnie tak to działa.

19. Jakość powietrza

Kolejną ważną sprawą może być monitorowanie powietrza. Wiedza o tym, czym oddychamy, ma bowiem kluczowe znaczenie dla naszego samopoczucia i zdrowia. Oczywiście są dostępne gotowe rozwiązania, ale możemy też zbudować własne, które będzie częścią naszego systemu. W przypadku tego projektu istotny jest czujnik jakości powietrza, który będzie mierzył stężenia pyłów $PM_{1,0}$, $PM_{2,5}$ i PM_{10} . Dostępnych jest kilka rozwiązań różniących się i ceną, i możliwościami. Pierwszym, w miarę tanim, rozwiązaniem może być czujnik Dust Sensor produkowany przez firmę Waveshare. Jest oparty na czujniku optycznym Sharp GP2Y1010AU0F, pozwalającym wykrywać cząstki o średnicy powyżej $0,8\text{ }\mu\text{m}$. Układ jest zasilany napięciem od 2,5 do 5,5 V i działa w zakresie do $500\text{ }\mu\text{g}/\text{m}^3$ z czułością $0,5\text{ V} / (100\text{ }\mu\text{g}/\text{m}^3)$. Można też użyć dokładniejszego czujnika, pozwalającego wykrywać cząstki o średnicy powyżej $0,3\text{ }\mu\text{m}$, a konkretnie: PSM3003. I to właśnie na nim się skupimy.

PSM3003 — charakterystyka i zasada działania

Czujnik ten umożliwia monitorowanie czystości powietrza, czyli ilości pyłu zawieszonego, który wdychamy. Mierzy on współczynniki $PM_{1,0}$, $PM_{2,5}$ oraz PM_{10} . Wbudowany sensor wykrywa już cząstki powyżej $0,3\text{ }\mu\text{m}$, np. kurz, dym papierosowy itp. Działanie czujnika opiera się na optycznej metodzie pomiaru zawartości pyłu zawieszonego. W tym

czujniku światło lasera pada na cząstki zanieczyszczeń, które przelatują przez komorę pomiarową. Ulega ono rozproszeniu i zostaje wykryte przez fotodetektor. Sygnał z niego jest następnie wzmacniany oraz poddawany filtracji i analizie przez wbudowany mikrokontroler. Następnie dane są wysyłane przez interfejs UART. Dla nas ważne jest też to, że układ pracuje z napięciem 3,3 V, więc nie musimy stosować dodatkowych układów konwertujących poziomy sygnałów.

Uproszczona specyfikacja PSM3003

- napięcie zasilania: 5 V,
- napięcie pracy: 3,3 V,
- pobór prądu: do 100 mA,
- pobór prądu w stanie spoczynku: do 200 μ A,
- zakres pomiarów:
 - od 0,3 μ m do 1,0 μ m,
 - od 1,0 μ m do 2,5 μ m,
 - od 2,5 μ m do 10 μ m,
- czas odpowiedzi: do 10 s,
- temperatura pracy: od -10°C do 60°C,
- wymiary płytki: 65 × 42 × 23 mm,
- masa: 36 g.

Pinout

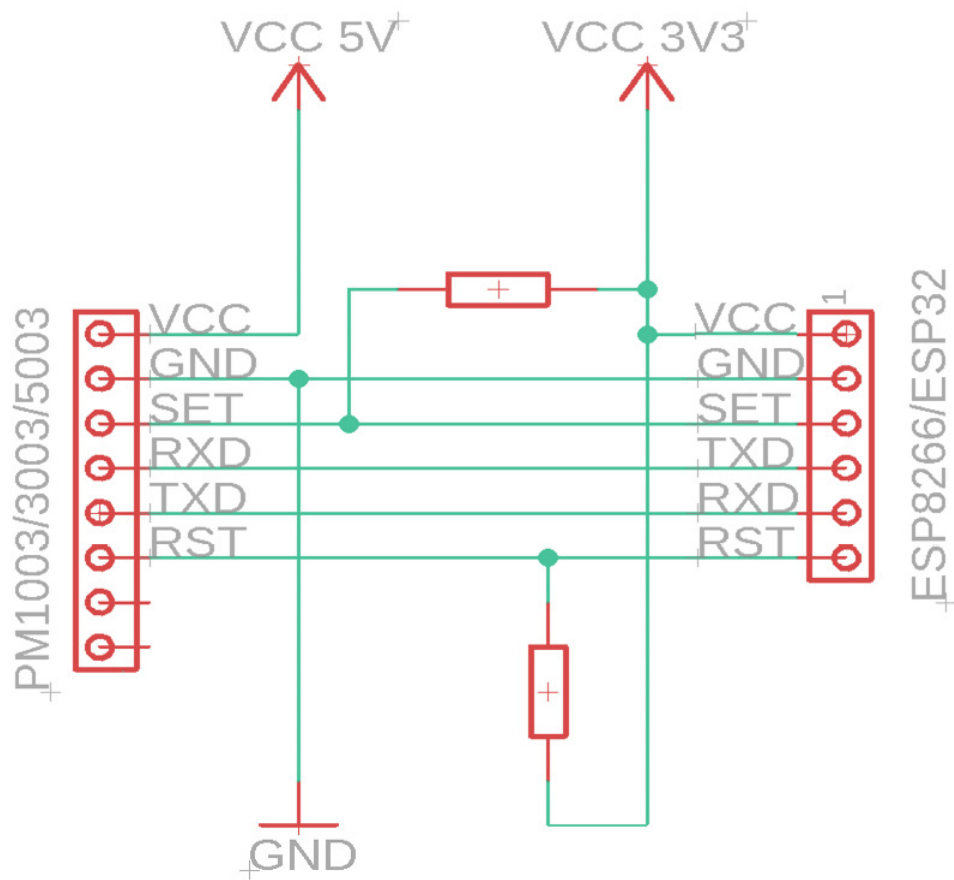
Moduł czujnika jest wyposażony w 8-pinowe złącze JST, na którym znajdziemy sygnały:

- pin 1 — VCC — zasilanie 5 V,
- pin 2 — masa urządzenia,
- pin 3 — SET — TTL 3,3 V,
- pin 4 — RXD — TTL 3,3 V,
- pin 5 — TXD — TTL 3,3 V,

- pin 6 — RESET — TTL 3,3 V,
- pin 7 — niepodłączony,
- pin 8 — niepodłączony.

Typowy schemat połączeń z mikrokontrolerem i komunikacja

Typowy schemat połączeń z mikrokontrolerem i komunikacja są pokazane na rysunku 19.1.



Rysunek 19.1. Schemat podłączenia PMS do mikrokontrolera⁷

Jak widać na schemacie, do komunikacji z czujnikiem będziemy używać kilku pinów i interfejsu UART. PMS3003 nadaje przy prędkości 9600 b/s, bez parzystości, z 1 bitem stopu.

Więcej danych znajdziecie w dokumentacji czujnika. Tu przytaczam tylko te podstawowe.

Kod programu

Nasz przykładowy kod będzie się opierał na tych samych rozwiązaniach co poprzednio. Komunikacja z czujnikiem będzie się odbywać na UART z użyciem pinów:

- *GPIO17* — TX,
- *GPIO16* — RX,
- *GPIO4* — SET,
- *GPIO5* — RESET.

Cały nasz kod prezentuje się następująco:

```
#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "freertos/semphr.h"
#include "esp_system.h"
#include "nvs_flash.h"
#include "esp_log.h"
#include "mqtt_client.h"
#include "driver/uart.h"
#include "driver/gpio.h"
```

```
#define UART_RX_PIN 16
#define UART_TX_PIN 17
#define SET_PIN 4
#define RESET_PIN 5
#define WIFI_SSID "SSID"
#define WIFI_PASSWORD "<hasło WiFi>"
#define MQTT_BROKER "<adres brokera MQTT>"
#define MQTT_PORT 1883
#define MQTT_TOPIC "Jakosc powietrza"
static esp_mqtt_client_handle_t mqtt_client;
static esp_err_t wifi_event_handler(void *ctx,
system_event_t *event) {
    switch (event->event_id) {
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect();
            break;
        case SYSTEM_EVENT_STA_CONNECTED:
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            esp_wifi_connect();
            break;
        default:
            break;
    }
    return ESP_OK;
}
```

```

static void wifi_init()
{
    tcpip_adapter_init();

    ESP_ERROR_CHECK(esp_event_loop_init(wifi_event_handler,
    NULL));

    wifi_init_config_t cfg =
    WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    wifi_config_t wifi_config = {
        .sta = {
            .ssid = WIFI_SSID,
            .password = WIFI_PASSWORD,
        },
    };
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));

    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA,
    &wifi_config));
    ESP_ERROR_CHECK(esp_wifi_start());
}

static esp_err_t
mqtt_event_handler(esp_mqtt_event_handle_t event) {
    switch (event->event_id) {
    case MQTT_EVENT_CONNECTED:
        ESP_LOGI("MQTT", "Połączono z brokerem MQTT");
        break;
    case MQTT_EVENT_DISCONNECTED:
        ESP_LOGI("MQTT", "Rozłączono z brokerem MQTT");
        break;
    }
}

```

```

        default:
            break;
    }
    return ESP_OK;
}

static void mqtt_init() {
    esp_mqtt_client_config_t mqtt_cfg = {
        .uri = "mqtt://" MQTT_BROKER ":" MQTT_PORT,
    };
    mqtt_client = esp_mqtt_client_init(&mqtt_cfg);
    esp_mqtt_client_register_event(mqtt_client,
    ESP_EVENT_ANY_ID, mqtt_event_handler, mqtt_client);
    esp_mqtt_client_start(mqtt_client);
}

void uart_init()
{
    uart_config_t uart_config = {
        .baud_rate = 9600,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .rx_pin = UART_RX_PIN,
        .tx_pin = UART_TX_PIN,
    };

    ESP_ERROR_CHECK(uart_param_config(UART_NUM_2,
    &uart_config));

    ESP_ERROR_CHECK(uart_set_pin(UART_NUM_2,
    UART_TX_PIN, UART_RX_PIN, UART_PIN_NO_CHANGE,
    UART_PIN_NO_CHANGE));
}

```

```

    ESP_ERROR_CHECK(uart_driver_install(UART_NUM_2,
1024, 0, 0, NULL, 0));
}
void gpio_init()
{
    gpio_config_t io_conf = {
        .pin_bit_mask = (1ULL << SET_PIN) | (1ULL <<
RESET_PIN),
        .mode = GPIO_MODE_OUTPUT,
        .intr_type = GPIO_INTR_DISABLE,
        .pull_up_en = 0,
        .pull_down_en = 0,
    };
    ESP_ERROR_CHECK(gpio_config(&io_conf));
}
void pms3003_reset()
{
    gpio_set_level(RESET_PIN, 0);
    vTaskDelay(100 / portTICK_PERIOD_MS);
    gpio_set_level(RESET_PIN, 1);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}
void app_main()
{
    nvs_flash_init();
    wifi_init();
    mqtt_init();
    uart_init();
}

```

```

    gpio_init();
        pms3003_reset();
    while (1) {
        uint8_t data[32];
        int len = uart_read_bytes(UART_NUM_2, data,
sizeof(data), 1000 / portTICK_RATE_MS);
        if (len == 32 && data[0] == 0x42 && data[1] ==
0x4D) {
            uint16_t pm1_0 = (data[6] << 8) | data[7];
            uint16_t pm2_5 = (data[8] << 8) | data[9];
            uint16_t pm10 = (data[10] << 8) | data[11];
            char message[64];
            sprintf(message, "{\"PM1.0\": %u,
\"PM2.5\": %u, \"PM10\": %u}", pm1_0, pm2_5, pm10);
            esp_mqtt_client_publish(mqtt_client,
MQTT_TOPIC, message, 0, 0, 0);
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

```

Omówienie elementów kodu

W kodzie widzicie znane Wam już bloki, takie jak definicje czy funkcje inicjujące WiFi i MQTT, oraz obsługę zdarzeń. Jednak doszły nam tu nowe rzeczy, które można pokrótce przedstawić i opisać dla porządku. Zatem do dzieła.

Funkcja `void uart_init()` odpowiada za inicjalizację modułu UART na mikrokontrolerze ESP32. Poniżej opisuję poszczególne kroki działania tej funkcji.

Najpierw konfigurujemy parametry UART:

```
uart_config_t uart_config =  
{  
    .baud_rate = 9600,  
    .data_bits = UART_DATA_8_BITS,  
    .parity = UART_PARITY_DISABLE,  
    .stop_bits = UART_STOP_BITS_1,  
    .rx_pin = UART_RX_PIN,  
    .tx_pin = UART_TX_PIN,  
};
```

Oto dalsze kroki w konfigurowaniu tych parametrów:

1. Tworzymy strukturę `uart_config_t`, która przechowuje konfigurację parametrów UART.
2. Ustawiamy prędkość transmisji (`baud_rate`) na 9600 b/s.
3. Konfigurujemy liczbę bitów danych (`data_bits`) na 8.
4. Wyłączamy kontrolę parzystości (`parity`).
5. Ustawiamy liczbę bitów stopu (`stop_bits`) na 1.
6. Określamy piny GPIO dla linii odbiorczej (`rx_pin`) i linii nadawczej (`tx_pin`).

```
ESP_ERROR_CHECK(uart_param_config(UART_NUM_2,  
    &uart_config));
```

Ustawiamy konfigurację parametrów modułu UART dla danego numeru UART (w tym przypadku używamy *UART2*). `uart_param_config` przyjmuje numer interfejsu UART i wskaźnik do struktury `uart_config_t` oraz ustawia parametry zgodnie z konfiguracją.

```
ESP_ERROR_CHECK(uart_set_pin(UART_NUM_2, UART_TX_PIN,  
    UART_RX_PIN, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE));
```

Ten wiersz odpowiada za ustawienie pinów dla linii nadawczej (*TX*) i odbiorczej (*RX*) modułu UART. `uart_set_pin` przyjmuje zaś numer interfejsu UART oraz numery pinów *TX* i *RX*, a także parametry zastępcze (`UART_PIN_NO_CHANGE`, co oznacza brak zmiany).

```
ESP_ERROR_CHECK(uart_driver_install(UART_NUM_2, 1024,  
0, 0, NULL, 0));
```

`uart_driver_install` przyjmuje numer interfejsu UART, rozmiar bufora odbiorczego (w tym przypadku 1024 bajty), parametry zastępcze oraz opcjonalne ustawienia, które w tym przypadku są ustawione na `NULL`.

Cała funkcja `uart_init` skupia się na konfiguracji i inicjalizacji modułu UART, by umożliwić mu poprawną komunikację z urządzeniami podłączonymi do tych linii transmisji szeregowej. Funkcja `ESP_ERROR_CHECK` sprawdza ewentualne błędy konfiguracji, co jest istotne w procesie debugowania i utrzymania systemu. Funkcję tę możecie swobodnie dodawać do wszelkich swoich programów, w których potrzebne może być zastosowanie komunikacji szeregowej. Tu warto wiedzieć, że *UART1* w urządzeniach z ESP8266 i ESP32 używamy do wgrywania oprogramowania i komunikacji z komputerem, np. w celu wymiany danych diagnostycznych, więc do innych urządzeń możemy używać *UART2* lub *UART3*.

Funkcja `gpio_init()` również opiera się na strukturze i w niej konfigurujemy piny GPIO jako wyjścia, które posłużą nam do obsługi sygnałów SET i RESET modułu PMS3003.

```
void gpio_init()  
{  
    gpio_config_t io_conf =  
    {  
        .pin_bit_mask = (1ULL << SET_PIN) | (1ULL <<  
RESET_PIN),
```

```

        .mode = GPIO_MODE_OUTPUT,
        .intr_type = GPIO_INTR_DISABLE,
        .pull_up_en = 0,
        .pull_down_en = 0,
    };
    ESP_ERROR_CHECK(gpio_config(&io_conf));
}

```

Poniżej znajdują się opisy poszczególnych pól konfiguracji pinów:

.pin_bit_mask

Pole to określa maskę bitową pinów GPIO, które zostaną skonfigurowane przez tę funkcję. W tym przypadku z użyciem operatora bitowego << tworzona jest maska, która ustawia bity na wartość 1 dla pinów o numerach SET_PIN i RESET_PIN.

.mode

Pole to określa tryb pracy pinów GPIO. W tym przypadku ustawiono go na GPIO_MODE_OUTPUT, co oznacza, że piny te będą działać jako wyjścia.

.intr_type

Pole to określa rodzaj przerwań obsługiwanych przez piny GPIO. W tym przypadku ustawiono je na GPIO_INTR_DISABLE, co oznacza brak obsługi przerwań dla tych pinów.

.pull_up_en oraz .pull_down_en

Te pola pozwalają na włączenie lub wyłączenie rezystorów podciągających (*pull-up*) lub opuszczających (*pull-down*) na pinach GPIO. W tym przypadku oba są ustawione na 0, co oznacza, że rezystory *pull-up* i *pull-down* są wyłączone.

Podsumowując, konfiguracja GPIO za pomocą tej struktury określa, że piny `SET_PIN` i `RESET_PIN` będą pracować jako wyjścia (`GPIO_MODE_OUTPUT`) bez obsługi przerwań, a rezystory *pull-up* i *pull-down* są wyłączone. W praktyce to podstawowa konfiguracja GPIO pozwalająca na ustawienie stanu *HI* lub *LOW*. Czyli wyjścia te mogą być używane do sterowania różnymi elementami, takimi jak diody LED, przekaźniki czy inne układy elektroniczne.

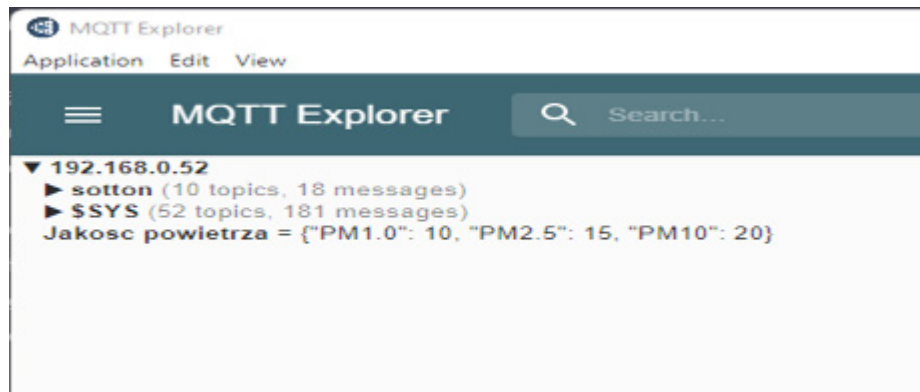
Funkcja `pms3003_reset()` — właśnie tu widzimy w praktyce to, o czym pisałem wyżej w omówieniu funkcji `gpio_init()`.

Na pin *RESET* czujnika PMS3003 wystawiamy stan niski (*LOW*), co spowoduje jego zresetowanie.

```
void pms3003_reset()
{
    gpio_set_level(RESET_PIN, 0);
    vTaskDelay(100 / portTICK_PERIOD_MS);
    gpio_set_level(RESET_PIN, 1);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}
```

Po krótkim czasie koniecznym na zresetowanie czujnika, czyli 100 ms, ustawiamy stan wysoki, *HI*. Tym samym czujnik nie zresetuje się samoczynnie. Oczywiście to bardzo prosta wersja, ale potrafi odczytać dane z czujnika i prezentuje je w temacie *Jakosc powietrza* na naszym brokerze w formacie JSON. Wiadomość jest pokazana na rysunku 19.2.

```
{"PM1.0": 10, "PM2.5": 15, "PM10": 20}
```



Rysunek 19.2. Dane z czujnika w formacie JSON

Interpretacja wyników

Wartości $PM_{1,0}$, $PM_{2,5}$ i PM_{10} , które odczytujemy z czujnika PMS3003, są wyrażone w mikrogramach na metr sześcienny ($\mu g/m^3$). Oto sposób, w jaki można interpretować te wartości:

- $PM_{1,0}$ — stężenie pyłów zawieszonych o średnicy mniejszej lub równej 1,0 mikrometra w 1 metrze sześciennym powietrza. To obejmuje zarówno drobiny $PM_{1,0}$, jak i większe.
- $PM_{2,5}$ — stężenie pyłów zawieszonych o średnicy mniejszej lub równej 2,5 mikrometra w 1 metrze sześciennym powietrza. $PM_{2,5}$ obejmuje drobiny $PM_{1,0}$, $PM_{2,5}$ i większe.
- PM_{10} — stężenie pyłów zawieszonych o średnicy mniejszej lub równej 10 mikrometrów w 1 metrze sześciennym powietrza. PM_{10} obejmuje drobiny $PM_{1,0}$, $PM_{2,5}$, PM_{10} i większe.

Wartości te są ważne w kontekście monitorowania jakości powietrza, ponieważ różne frakcje pyłów mogą mieć różny wpływ na nasze zdrowie i samopoczucie. Na przykład

drobiny $PM_{2,5}$ mogą przenikać głęboko do płuc, co może prowadzić do problemów zdrowotnych, zwłaszcza u osób cierpiących na choroby układu oddechowego.

Ogólnie rzecz biorąc, niższe wartości $PM_{1,0}$, $PM_{2,5}$ i PM_{10} są pożądane, ponieważ oznaczają niższe stężenie pyłów zawieszonych i lepszą jakość powietrza. Standardy jakości powietrza określają dopuszczalne limity dla tych parametrów, a przekroczenie tych limitów może być szkodliwe dla zdrowia. Ważne jest monitorowanie i utrzymanie tych wartości na odpowiednim poziomie, zwłaszcza w obszarach miejskich czy przemysłowych, gdzie może występować zwiększone zanieczyszczenie powietrza.

W naszym przypadku mamy np. $PM_{1,0} = 10 \mu g/m^3$, czyli stężenie pyłów zawieszonych o średnicy mniejszej lub równej 1,0 mikrometra wynosi 10 mikrogramów na metr sześcienny powietrza. Czy jest to dużo, czy mało? Musimy się odnieść do norm określających jakość powietrza.

Orientacyjne kategorie stężeń pyłów $PM_{2,5}$ wg norm WHO (Światowa Organizacja Zdrowia) i EPA (Agencja Ochrony Środowiska w USA) wynoszą:

- **dobre** — poniżej $12 \mu g/m^3$ (WHO) lub poniżej $12,0 \mu g/m^3$ (EPA),
- **akceptowalne** — $12 - 35,4 \mu g/m^3$ (WHO) lub $12,1 - 35,4 \mu g/m^3$ (EPA),
- **niezdrowe dla osób wrażliwych** — $35,5 - 55,4 \mu g/m^3$ (WHO) lub $35,5 - 55,4 \mu g/m^3$ (EPA),
- **niezdrowe** — $55,5 - 150,4 \mu g/m^3$ (WHO) lub $55,5 - 150,4 \mu g/m^3$ (EPA),
- **bardzo niezdrowe** — $150,5 - 250,4 \mu g/m^3$ (WHO) lub $150,5 - 250,4 \mu g/m^3$ (EPA),
- **zagrożenie dla zdrowia** — powyżej $250,5 \mu g/m^3$ (WHO) lub powyżej $250,4 \mu g/m^3$ (EPA).

Oceniając swoje wyniki, możesz porównać je z tymi normami. Jeśli Twoje wyniki są poniżej tych wartości, to jakość powietrza można uznać za zgodną z normami. Jeśli natomiast przekraczają normy, oznacza to potencjalne zagrożenie dla zdrowia publicznego i mogą być konieczne działania mające na celu poprawę jakości powietrza.

Warto pamiętać, że wartości graniczne mogą się zmieniać. Aktualne normy i wytyczne mogą być dostępne w dokumentach Ministerstwa Klimatu i Środowiska lub innych organów odpowiedzialnych za środowisko. Zalecam również śledzenie aktualizacji norm i wytycznych dotyczących jakości powietrza.

[Z](#) Źródło: Schemat stworzony w narzędziu Eagle; dostęp 11.06.2024.

Zakończenie

Ten ostatni rozdział zarazem otwiera przed Wami nowe perspektywy. Jest jak finał udanego spektaklu, który z jednej strony kończy pewien etap, z drugiej zaś uchyla drzwi do nieskończonych możliwości. Książka ta stała się dla Was nie tylko zbiorem informacji o tych niewielkich, lecz potężnych modułach, ale także przewodnikiem po świecie tworzenia i projektowania układów w dziedzinie Internetu Rzeczy.

W miarę jak razem eksplorowaliśmy kolejne zakamarki labiryntu i zmagaliśmy się z programowaniem naszych urządzeń oraz integrowaniem ich z różnymi sensorami, a także rozwijaliśmy umiejętności na polu komunikacji bezprzewodowej, powoli zdawaliście sobie sprawę, że te małe układy mikrokontrolerowe otwierają przed Wami furtkę do budowy pełnych potencjału urządzeń, dzięki którym tak wiele można osiągnąć w projektowaniu rozwiązań IoT.

Przez te wszystkie strony wspólnie odkrywaliśmy, jak za pomocą tych niewielkich układów zrealizować rozbudowane systemy monitoringu parametrów oraz automatyki domowej, które są związane z Internetem Rzeczy. Zdobytą wiedzę możecie teraz wykorzystać do stworzenia projektów, które nie tylko zaspokoją Wasze własne potrzeby, ale także zaimponują profesjonalistom.

Dziękuję Wam, drodzy Czytelnicy, za to, że towarzyszyliście mi w tej podróży. Teraz, z pełnym zestawem umiejętności, można by rzec, fundamentalnej wiedzy, czas ruszyć dalej w świat projektowania Internetu Rzeczy na profesjonalnym poziomie. By technologia była nie tyle z nami, ile dla nas

i wspomagala nasze codzienne zycie w taki sposob, w jaki to zaplanowalismy.

Zamykam tę książkę z uśmiechem na twarzy i podziękowaniem w sercu. Mam nadzieję, że ten spacer był dla Was inspiracją, a zdobyta tutaj wiedza pomoże Wam w tworzeniu nowoczesnych, innowacyjnych rozwiązań. Do zobaczenia w świecie bezgranicznych możliwości i efektów, które tylko czekają, abyście je odkryli!

PROGRAM PARTNERSKI

— GRUPY HELION —

- 
1. ZAREJESTRUJ SIĘ
 2. PREZENTUJ KSIĄŻKI
 3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Spis treści

Wstęp

Czego możesz się spodziewać

Dlaczego ESP8266 i ESP32?

Dla kogo jest ta książka?

CZĘŚĆ I. Podstawy świata IoT

1. Trochę historii czy wprowadzenie?

2. Sprzętowisko — czyli z czym po drodze

ESP8266 — charakterystyka układu

ESP32 — im dalej w las...

ESP-PROG — gwarancja dobrej zabawy i bezpieczeństwa

Pamięci flash w układach ESP

3. Interfejsy i peryferia

Digital GPIO

Analog GPIO

Napięcie referencyjne ADC

Konfigurowalność

ADC

UART

1-Wire

I2C

Podsumowanie

4. Bezpieczeństwo

Hipotetyczne rozważania

FuseBity

Szyfrowanie pamięci flash

Secure Boot

5. Środowisko

CZĘŚĆ II. Przygotowania do programowania

6. Arduino IDE

Omówienie struktury programu w Arduino

7. Visual Studio Code i PlatformIO

Tworzymy projekt

8. Visual Studio Code — ESP-IDF

Pierwszy program w ESP-IDF

Omówienie programu w ESP-IDF

Kompilacja i uruchomienie programu

Podsumowanie

CZĘŚĆ III

Wejście do labiryntu

9. Czym jest IoT

10. Protokół MQTT

11. Broker

Brokery dostępne online — wady i zalety

Mosquitto — opensource'owy bezpłatny broker

Narada MQTT — broker dla Androida

12. Stawiamy własny lokalny broker MQTT

Przydatne narzędzia do monitorowania brokera

Przydatne dane systemowe

JSON — jak go zrozumieć

JSON — czytanie danych

CZĘŚĆ IV. Do dzieła!

13. Pierwszy projekt

Przebudzenie — wariant pierwszy

Przebudzenie — wariant drugi

Przebudzenie — wariant trzeci

14. Sterowanie urządzeniami

Dlaczego bezpieczeństwo jest priorytetem?

Kilka zasad bezpieczeństwa przy sterowaniu urządzeniami

Izolacja elektryczna

Przerwy w zasilaniu

Znajomość urządzeń

Używaj odpowiednich narzędzi

Sterowanie urządzeniami za pomocą mikrokontrolerów

Separacja zasilania

Zabezpieczenia przeciwprzepięciowe

Ostrzeżenia przed wysokim napięciem

Podsumowanie

[Schemat połączeń](#)
[Oprogramowanie sterownika](#)
[15. Ekoelektronika](#)
[Budujemy WallPanel](#)
[WallPanel — wariant drugi](#)
[16. Dashboard](#)
[17. Sterowanie ogrzewaniem, na przykład w akwarium](#)
[Instalacja sterownika DS18B20](#)
[Wersja pierwsza](#)
[Wersja druga](#)
[Kod kontrolera](#)
[Omówienie kodu](#)
[18. Monitorowanie zużycia energii](#)
[Kod programu](#)
[Omówienie kodu](#)
[19. Jakość powietrza](#)
[PSM3003 — charakterystyka i zasada działania](#)
[Uproszczona specyfikacja PSM3003](#)
[Pinout](#)
[Typowy schemat połączeń z mikrokontrolerem i komunikacja](#)
[Kod programu](#)
[Omówienie elementów kodu](#)
[Interpretacja wyników](#)
[Zakończenie](#)