Instytut Informatyki Politechniki Śląskiej

Zespół Mikroinformatyki i Teorii Automatów Cyfrowych

Rok akademicki:	Rodzaj studiów*: SSI/NSI/NSM	Przedmiot (Języki Asemblerowe/SMiW):	Grupa	Sekcja
2018/2019	**SSI**	**SMiW**	**5**	**9**

Imię:	**Bartłomiej**	**Prowadzący**: OA/JP/KT/GD/BSz/GB	**JP**
Nazwisko:	**Krasoń**		

Raport końcowy

Temat projektu:

Broker MQTT na Intel Edison

Data oddania: **dd/mm/rrrr**	**11/02/2019**

Spis treści

Temat projektu

Tematem projektu było postawienie brokera (serwera) obsługującego protokół MQTT na mikrokontrolerze jakim jest Intel Edison, który magazynuje przesyłane do niego dane z lokalnej sieci.

Opis założeń

Założenia projektu są następujące:

- broker działa w lokalnej sieci Wi-Fi
- broker pobiera dane wysyłane do niego przez protokół MQTT oraz magazynuje je lokalnie
- możliwe jest wysyłanie danych do ”góry” – przyjęte zostało wysyłanie danych na chmurę (Google Drive)
- zapewnienie bezpieczeństwa przez certyfikację SSL
- zapewnione zasilanie podtrzymujące (bateria + usb)

Funkcje urządzenia

Urządzenie po zasileniu uruchamia program działający cały czas w tle tzw. demon, który odpowiada za lokalne zapisywanie wysyłanych do niego danych przez protokół MQTT. Po przesłaniu wiadomości ”send” w temacie ”conf” następuje przesłanie danych na Google Drive, obecnie na konto: **edison.mqtt.data@gmail.com**, jednakże folder na który przesyłane są dane, może być współdzielony z każdym innym użytkownikiem platformy Google Drive.

Analiza zadania

Software

Założeniem projektu jest zbieranie danych wysyłany na urządzenie przy pomocy protokołu MQTT oraz zapewnienie przesyłania ich ”w górę”. W technologii MQTT za zadanie pobierania danych odpowiada tzw. broker, który tworzy host na swoim IP. Natomiast odczytywanie wysyłanych danych na podanego brokera (hosta) realizuje tak de facto klient subskrybujący tego brokera. Także cel zadania sprowadził się do postawienia brokera na Edisonie, przy wykorzystaniu gotowej, open-sourcowej jego implementacji oraz napisaniu programu klienta, który dla postawionego brokera, subskrybuje wszystkie jego tematy oraz magazynuje lokalnie dane, jakie są mu wysyłane, przy czym odpowiednio reaguje na różne polecanie wysyłane do niego, m. in. umożliwiające przesyłanie danych na chmurę. Następnie trzeba było zapewnić aby program ten działał cały czas w tle, gdy Edison jest uruchomiony, możliwe to było dzięki wykorzystaniu serwisów, które w technologii Linuxa odpowiadają za uruchamianie usług (demonów) na systemie. Wybór Google Drive jako technologii chmurowej podyktowany jest tym, ze jest to najpopularniejsza obecnie platforma do przechowywania danych ”w chumrze”. Dostęp do niej jest prosty i bezpłatny. Google udostępnia również API, które służy do kontaktu machine-machine jeżeli chodzi o przesyłanie danych na platformę, co było konieczne aby móc zrealizować to zadanie. Wykorzystanie open-sourcowego programu **Gdrive** umożliwiło realizację komunikacji z platformą przy wykorzystaniu poleceń konsolowych.

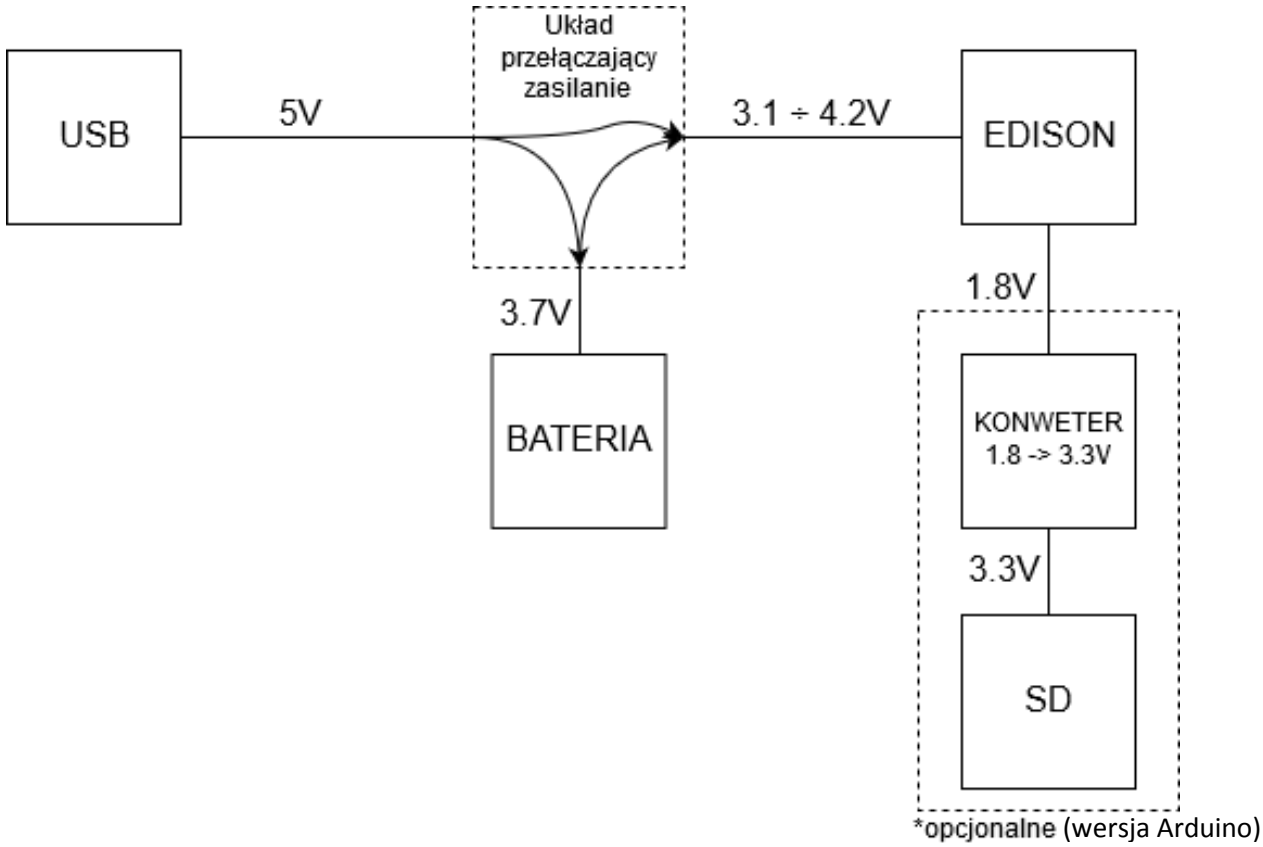
Hardware

Realizacja zadania opierała się na wykorzystaniu Intel Edisona, który sam w sobie już jest modułem, jednakże nie udostępniającym wystarczających interfejsów. W tym celu wykorzystano moduł rozszerzający jakim jest Intel Edison Mini Breakout-board. Udostępnia on zminimalizowany interfejs zapewniający zasilanie układu oraz możliwość podłączenia go do komputera-hosta, umożliwiając przy tym wgranie na niego nowego obrazu systemu oraz pełną jego obsługę. W celu rozwojowym skorzystano również z drugiej, większej płytki rozwojowej Intel Edison Arduino Breakout Kit. Do niej zaprojektowano płytkę umożliwiającą podłączenie do modułu pamięci FRAM w celu wydłużenia żywotności całego układu.

Specyfikacja wewnętrzna urządzenia

Schemat blokowy urządzenia

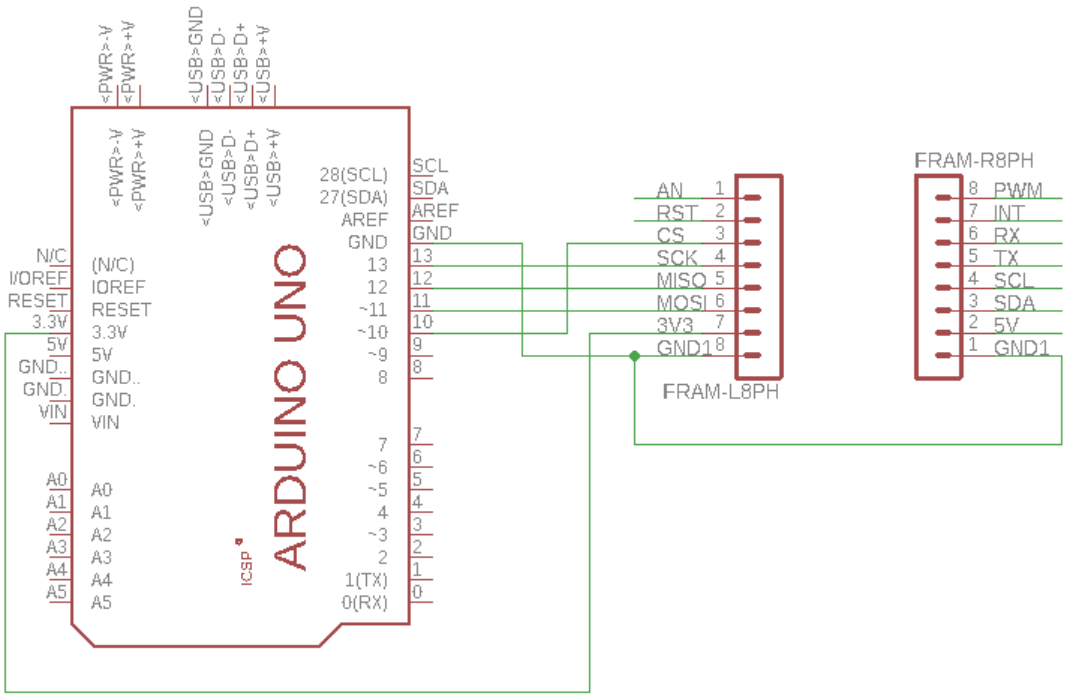
(płytki rozwojowej)



*opcjonalne (wersja Arduino)

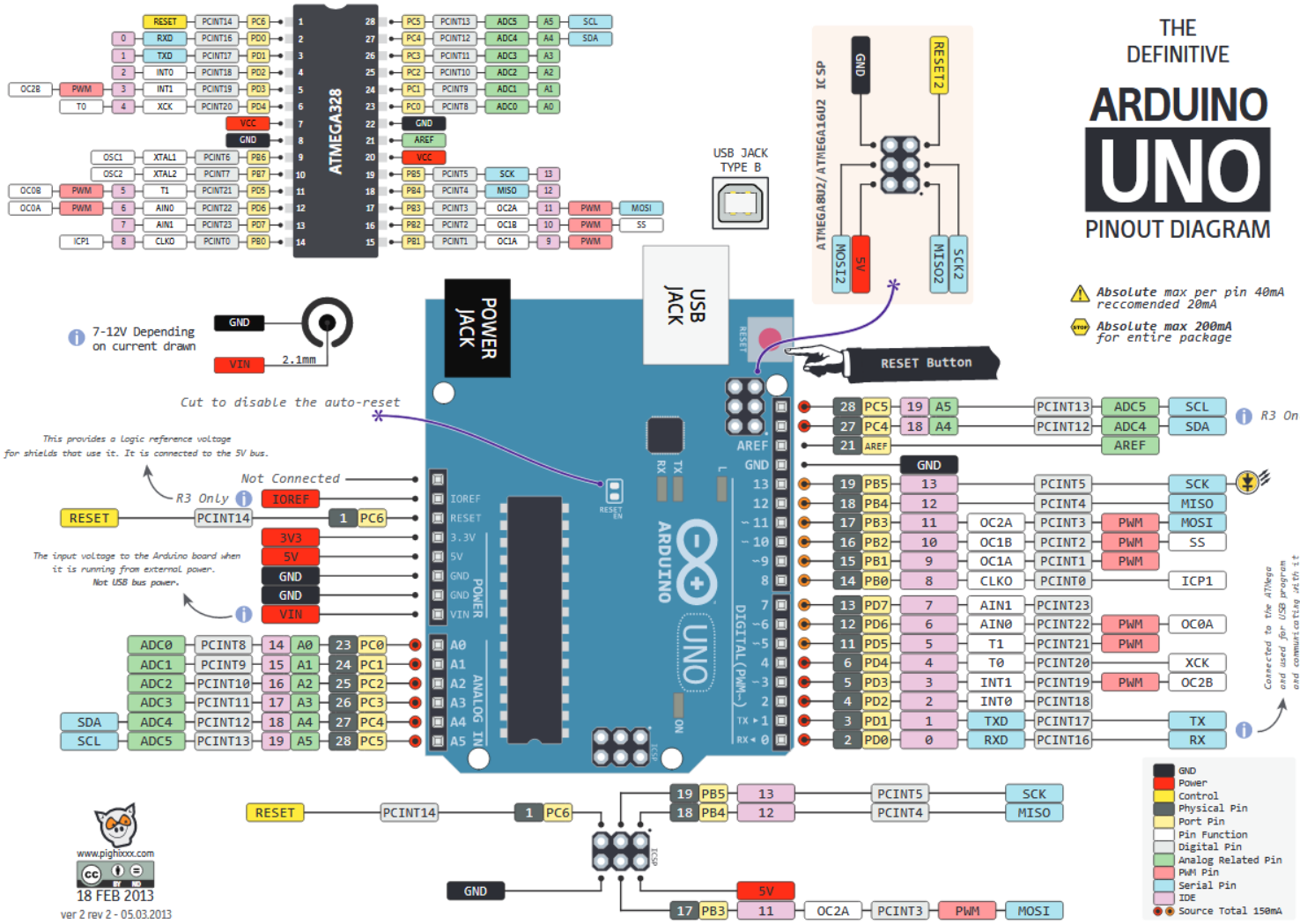
Schemat ideowy urządzenia

(podłączenie modułu pamięci FRAM do większego Kitu)



Opis funkcji bloków układu

- Intel Edison Arduino Breakout Kit – jest to większa płytka rozwojowa dla Edisona. Jej cały format (wyjścia, rozmiary itp.) zgodne są ze standardem Arduino Uno. Moduł ten umożliwia m.in. komunikację przez interfejs SPI z Edisonem, którą wykorzystujemy przy podłączeniu modułu pamięci FRAM. Wszystkie udostępnione wyjścia płytki są podciągane do 3.3V (oryginalnie sam Edison wystawia 1.8V). Moduł zawiera w sobie również 2 złącza USB, które umożliwiają zasilanie układu oraz podłączenie go do komputera-hosta z którego wygrywa się oprogramowanie. Szczegółowy schemat wyprowadzeń:



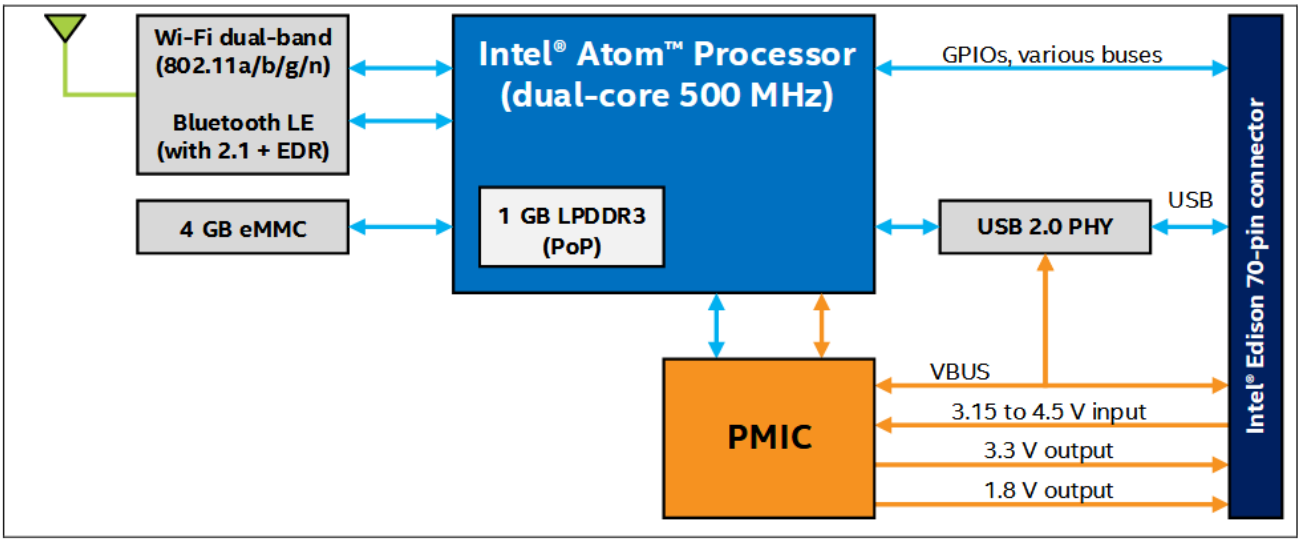
Źródło: http://marcusjenkins.com/arduino-pinout-diagrams/

- Moduł pamięci FRAM CY15B104Q – będzie przechowywał dane lokalnie, do czasu przesłania ich na chmurę. Zdecydowano się na podłączenie pamięci FRAM z powodu jej niezawodności jeżeli chodzi o lokalne przechowywanie danych w układzie. Pamięć FRAM góruje nad wmontowaną w moduł Edisona pamięcią RAM pod względem żywotności, która może dokonać nawet do 100 trylionów odczytów/zapisów. Jedynym minusem jest jej rozmiar, bo wynosi zaledwie 4MB, gdy pojemność wbudowanej pamięci Flash Edisona wynosi 4GB.

- Intel Edison – moduł zawierający:
    - procesor Intel Atom dual-core, dual-threaded 500MHz, z mikrokontrolerem Intel Quark 100MHz
    - pamięć RAM 1GB LPDDR3 POP memory
    - pamięć wewnętrzną 4GB eMMC Flash – w technologii manager NAND (z rozszerzeniem wear-leveling zapewniającym znaczne wydłużenie żywotności pamięci)
    - zasilanie TI SNB9024 zarządzanie zasilaniem IC
    - moduł WI-Fi stadard IEEE 802.11 a/b/g/n
    - Bluetooth 4.0 + 2.1 EDR
    - złącze 70-pinowe Hirose DF40 Series
    - złacze USB 2.0 – kontroler OTG
    - 2 zegary 19.2 MHz i 32 kHz
    - 40 wejść/wyjść ogólnego przeznaczenia w tym:
        - interfejs karty SD
        - 2 kontrolery UART (1 pełny, 1 szeregowy Rx/Tx)
        - 2 kontrolery I^2C
        - 1 kontroler SPI (2 x CS)
        - kontroler I2S
        - 14 dodatkowych pinów GPIO

Moduł ten posiada wgrany system operacyjny jakim jest Linux Yocto 3.5. Na nim przechowywane są programy stanowiące główny problem projektu oraz dane, które następnie przesyłane są na chmurę.

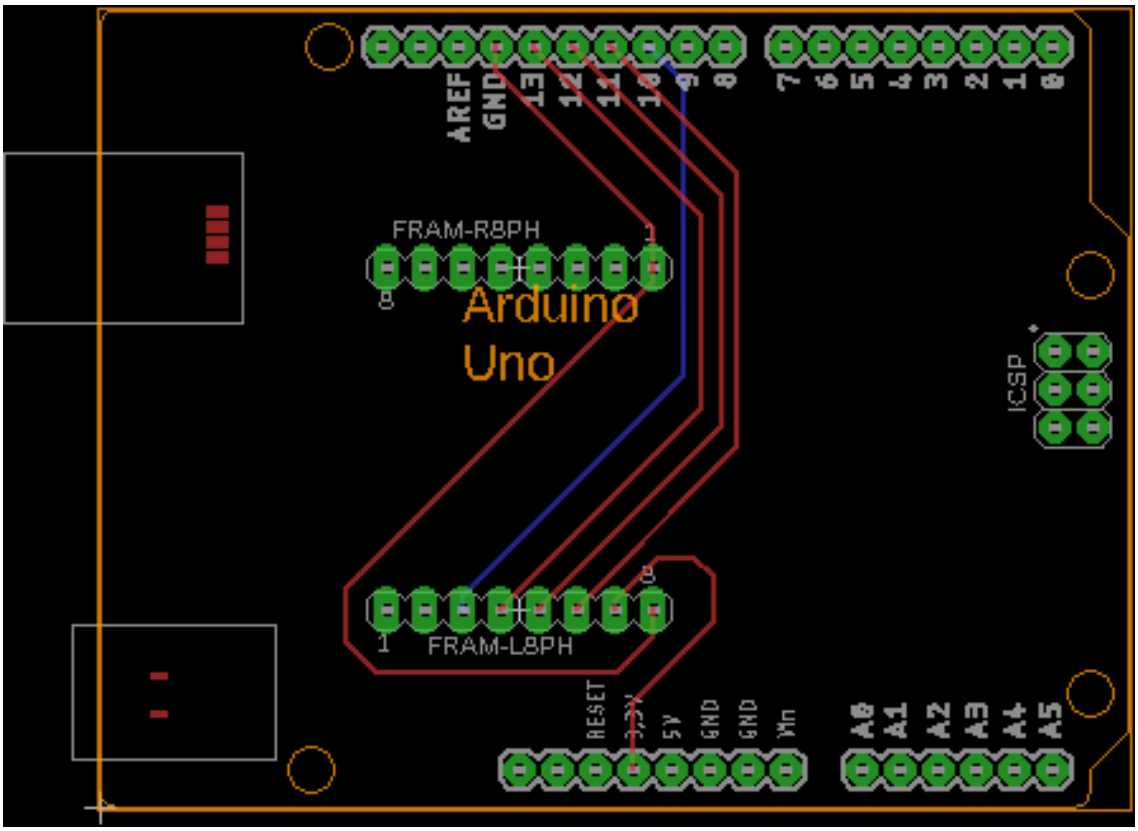
Schemat blokowy modułu Edisona:



Źródło: https://www.intel.com/content/dam/support/us/en/documents/edison/sb/edison-module-hardware-guide.pdf

Schemat montażowy PCB

(podłączenie modułu pamięci FRAM do większego Kitu)



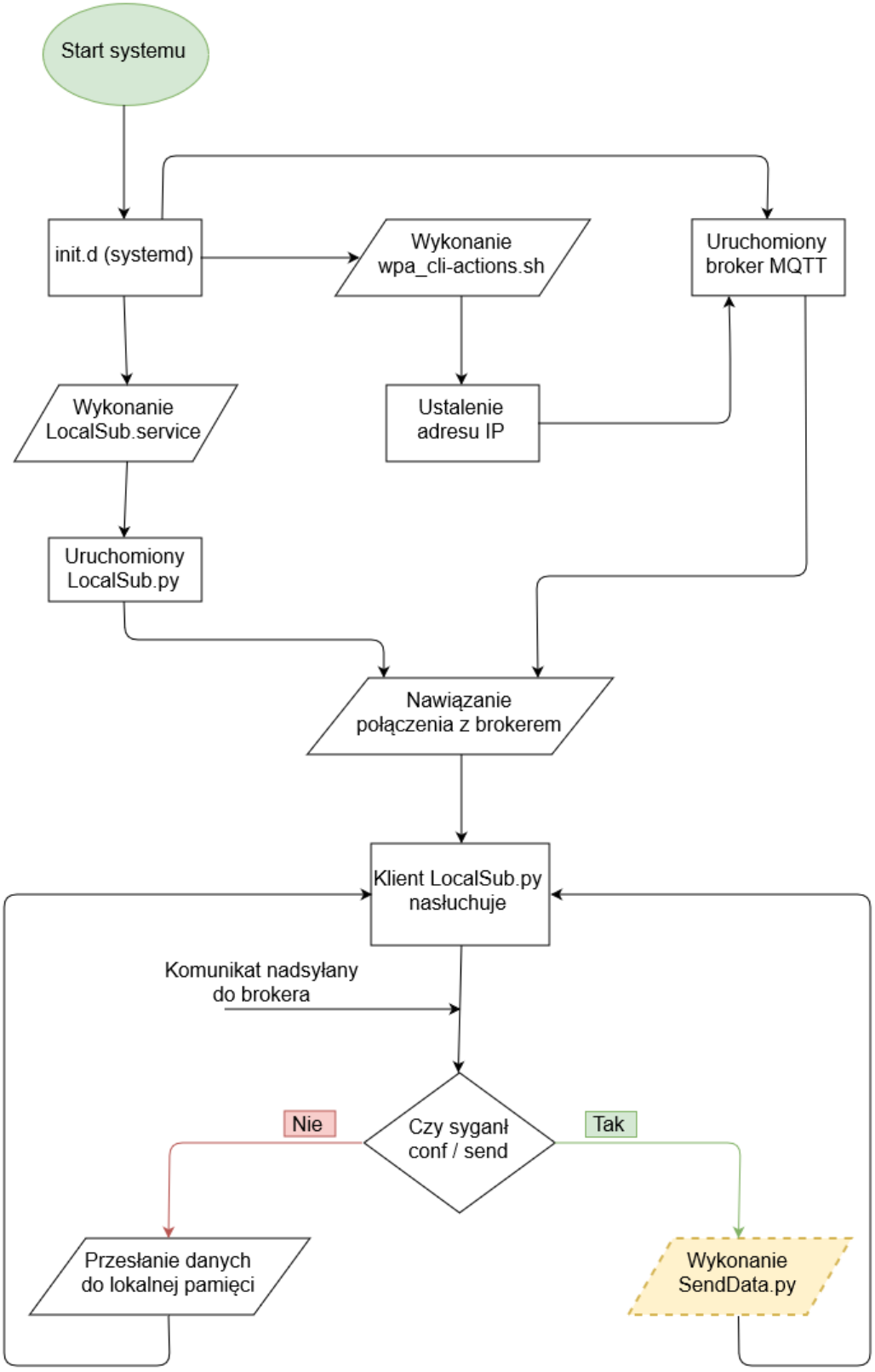
Lista elementów

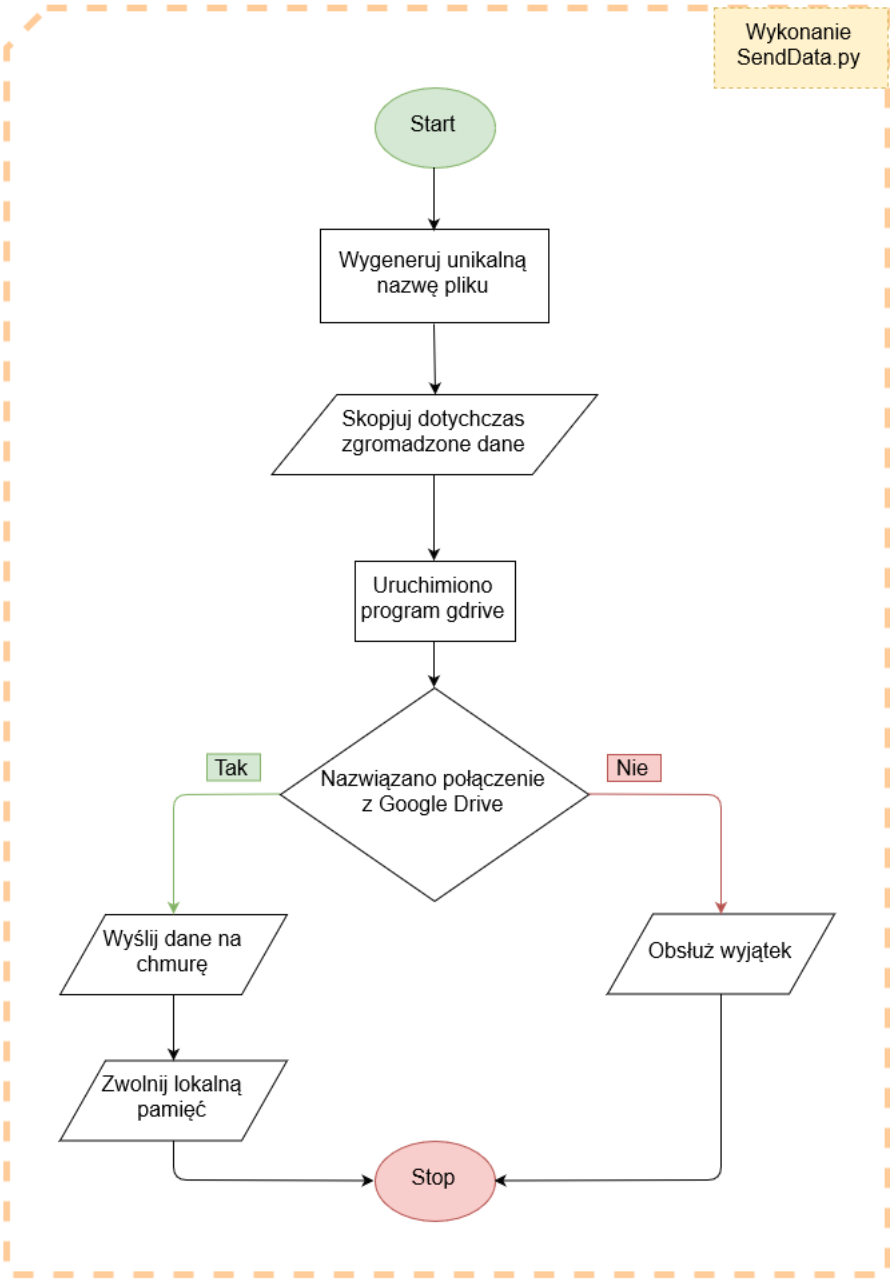
L.P.	Oznaczenie	Nazwa Elementu	Obudowa	Producent	Oznaczenie producenta	Dostawca	Oznaczenie dostawcy
1	Arduino Uno	Intel Edison Arduino Breakout Kit	standard Arduino Uno	Intel	ARDUINO.AL.B	SparkFun	DEV-13097
2	FRAM	FRAM 2 click	mikroBUS	MikroElektronika	MIKROE-2768	MikroElektonika	MIKROE-2768
3	-	Intel Edison	złącze Hirose DF40 Series	Intel	EDI1.SPON.AL.S	SparkFun	DEV-13024
4	-	Intel Edison Mini Breakout Kit	złącze Hirose DF40 Series	Intel	-	SparkFun	DEV-13025

Oprogramowanie

Oprogramowanie projektu jest jego głównym problemem, stąd jest ono nieco bardziej złożone. Nie stanowi go pojedynczy program, a kilka programów współpracujących ze sobą, coś na podobę działania mikroserwisów. Większość oprogramowania napisana jest w języku Python, gdyż system Linux Yocto ma wbudowany jego interpreter oraz jest to jeden z lepiej współpracujących języków programowania właśnie z tym systemem.

Schemat blokowy współdziałania oprogramowania





Opis poszczególnych programów

- **LocalSub.py** – program stanowiący klienta w protokole MQTT subskrybującego wszystkie tematy naszego lokalnego brokera. Po pierwsze odpowiada za nawiązanie połączenia z brokerem postawionym w lokalnej sieci. Broker korzysta z certyfikowanego połączenia SSL, co wymagało wygenerowania plików certyfikujących oraz odpowiednie podpięcie ich do programu. Po drugie odbiera dane przesyłane na brokera w postaci komunikatów MQTT oraz zapisuje je w pamięci wewnętrznej Edisona (bądź pamięci FRAM). Po trzecie reaguje odpowiednio na specyficzne sygnały przesyłane w postaci komunikatów MQTT. Na chwilę obecną obsługiwany jest komunikat ”send” wysłany w temacie ”conf” powodujący przesłanie danych na chmurę.
- **LocalSub.service** – serwis odpowiadający za uruchamianie programu LocalSub.py w trybie demona. Dzięki niemu, program ten może działać cały czas w tle, automatycznie uruchamiając się po podłączeniu zasilania do Edisona i prawidłowym nawiązaniu połączenia z lokalną siecią Wi-Fi.
- **SendData.py** – program odpowiadający za wysyłanie danych na chmurę. Uruchamiany przez program LocalSub.py, gdy wykryje on odpowiedni komunikat wymuszający przesłanie danych na chmurę. Program ten wysyła plik z dotychczas zgromadzonymi danymi w pamięci wewnętrznej,

ustala nazwę przesyłanego pliku na unikalną – wyznaczoną przez aktualny czas z dokładnością do 1 s. Gdy przesłanie danych się powiedzie, zwalnia on zajmowaną pamięć.

- **gdrive** – program służący do zdalnej komunikacji typu machine-machine, z dyskiem Google Drive. Udostępnia on konsolowy interfejs komunikacyjny, który wykorzystywany jest przez program SendData.py Program korzysta z certyfikowanego połączenia O-Auth 2.0, co wymagało wygenerowania odpowiednich plików certyfikujących i umieszczenie ich w pamięci Edisona oraz odpowiednie podpięcie ich do programu.

Opis plików konfiguracyjnych

- **mosquitto.conf** – plik konfiguracyjny brokera uruchamianego na Edisonie. Lista istotnych zmiennych konfiguracyjnych:
    - **port 8883** – określa na jakim porcie będzie działał host brokera w sieci. Port 8883 nie jest przypadkowy, gdyż jest on zalecanym portem, gdy stosuje się dla brokera certyfikację TLS/SSL,
    - **cafile /etc/mosquitto/certs/ca.crt** – wymagana gdy stosuję się SSL, wskazuje skąd broker ma pobierać certyfikat CA, w celu porównania go z certyfikatem klienta,
    - **certfile /etc/mosquitto/certs/server.crt** – wymagana przez SSL, określa ścieżkę do certyfikatu serwera,
    - **keyfile /etc/mosquitto/certs/server.key** – wymagana przez SSL, określa ścieżkę do klucza generowanego na potrzeby certyfikatu serwera,
    - **tls_version tlsv1** – określa wersję TLS, z której ma korzystać certyfikacja TLS/SSL.
- **/etc/wpa_supplicant/wpa_cli-actions.sh** – plik odpowiedzialny za ustalenie adresu IP Edisona w sieci. Odpowiednie wyedytowanie tego pliku pozwala na ustalenie z góry adresu hosta naszego Edisona, przy wykorzystaniu polecenia ifconfig.

Opis plików certyfikujących

Zestaw potrzebnych plików:

- Brokera:
    - **server.key** – klucz certyfikatu brokera, potrzeby do utworzenia pliku server.csr
    - **server.csr** – plik żądania certyfikatu, potrzeby do utworzenia pliku server.crt
    - **server.crt** – właściwy plik certyfikujący brokera
- Klienta:
    - **ca.key** – klucz certyfikatu klienta, potrzebny do utworzenia pliku ca.crt

ca.crt – właściwy plik certyfikujący klientów

Opis interakcji oprogramowania z układem elektronicznym

Po podaniu zasilania do układu, rozpoczyna się *booting.* Uruchamiany jest wewnętrzny kernel, po czy uruchamiani się Linux. Następuje inicjalizacja pierwszego procesu init.d (systemd), który następnie odpala pozostałe usługi systemowe. W tym miedzy innymi uruchamia nasz program LocalSub.py, dzięki zastosowaniu serwisu LocalSub.service, co powoduje, że po podaniu zasilania Edison, automatycznie – po uruchomieniu systemu – będzie odczytywał wysyłane na jego brokera dane w protokole MQTT. Jeżeli do układu podłączy się baterię (umożliwiają to obie płytki rozwojowe: Mini Breakout Kit oraz Arduino Breakout Kit) po odłączeniu głównego zasilania (5V z USB), Edison nadal będzie działa, dzięki zastosowaniu ”układu przełączającego zasilanie”. Układ ten pełni również rolę doładowywania baterii, w trakcie gdy dostarczane jest zasilanie z USB, co powoduje znaczne wydłużenie żywotności całego układu oraz jego niezawodność, w trakcie zaniku, głównego źródła zasilania.

Specyfikacja zewnętrzna

Urządzenie będące tematem mojego projektu jest tak jakby mini-komputerem osobistym, a to za sprawą tego, że wgrany jest na nim system operacyjny - Linux Yocto. Brak w tym urządzeniu jest mechanizmów sterujących, czy innych elementów wykonawczych, gdyż główną rolą działania tego urządzenia jest pośredniczenie w bezprzewodowym przesyle danych pomiędzy lokalną siecią czujników a bazą danych ”w chmurze”. Istotą tego projektu jest zestaw oprogramowania wgranego na tym mikrokontrolerze – umożliwiający wspomniany przesył danych.

Opis reakcji na zdarzenia zewnętrzne

Tak jak już wcześniej było wspominane, program uruchomiony na Edisonie reaguje na wszystkie komunikaty przesyłane na brokera postawionego na jego hoście w protokole MQTT. Sposób reakcji na wysyłane komunikaty opisuje schemat w rozdziale: Oprogramowanie.

Dodatkowo ”Intel Edison Mini Breakout Kit” jest wyposażony w tzw. układ ”battery backup” – czyli układ podtrzymujący zasilanie. Układ zasilania tej płytki, składa się z źródła 5V z USB oraz 3.7V z baterii Li-Ion. Schemat: Schemat blokowy urządzenia. Układ ten działa w taki sposób, że gdy dostarczane jest zasilanie z UBS, odpowiednie napięcie jest podawane na zasilanie Edisona, a resztą doładowywana jest bateria. Natomiast gdy nastąpi odłączenie układu od zasilania z USB, ten zaczyna być zasilany baterią, tak że układ pozostaje cały czas zasilany, nawet przy zaniku jego głównego źródła zasilania.

Skrócona instrukcja obsługi urządzenia

Gdy chcemy tylko uruchomić Edisona, oraz zainstalowany na nim system operacyjny wraz z jego wszystkimi serwisami, wystarczy podłączyć go do zasilania, po czym system sam się uruchamia. Złącza opisane w rozdziale: Złącza układu.

Natomiast gdy chcemy umożliwić bezpośrednio pracę z systemem operacyjnym, należy podłączyć jego płytkę deweloperską do komputera na którym znajduje się system hostowy. Na początku moduł należy podłączyć dwoma kablami micro USB typu B – zapewniające komunikację szeregową przez terminal oraz zasilanie i transfer danych. Nawiązać połączenie można m. in. przy wykorzystaniu programu PuTTy.

Natomiast co jest istotnym celem projektu, to odbieranie danych wysyłanych na jego hosta w protokole MQTT. Prezentuję wszystkie parametry jaki musi podać klient publikujący, aby mógł nawiązać takie połączenie:

- **Adres hosta** – adres IP Edisona (domyślnie: 192.168.43.250)
- **Temat** – przesył danych na dowolnym, konfiguracja na *”conf”*
- **Port** – 8883
- **Plik certyfikujący** – ścieżka do piku certyfikującego *”ca.crt”* (domyślne pliki certyfikujące na GitLabie /Software/MQTTBridgeClient/certs/keysHS)
- **Widomość** – treść komunikatu np. dane do przesłania

Przykład wykorzystania klienta mosquitto do pobrania z: https://mosquitto.org/download/

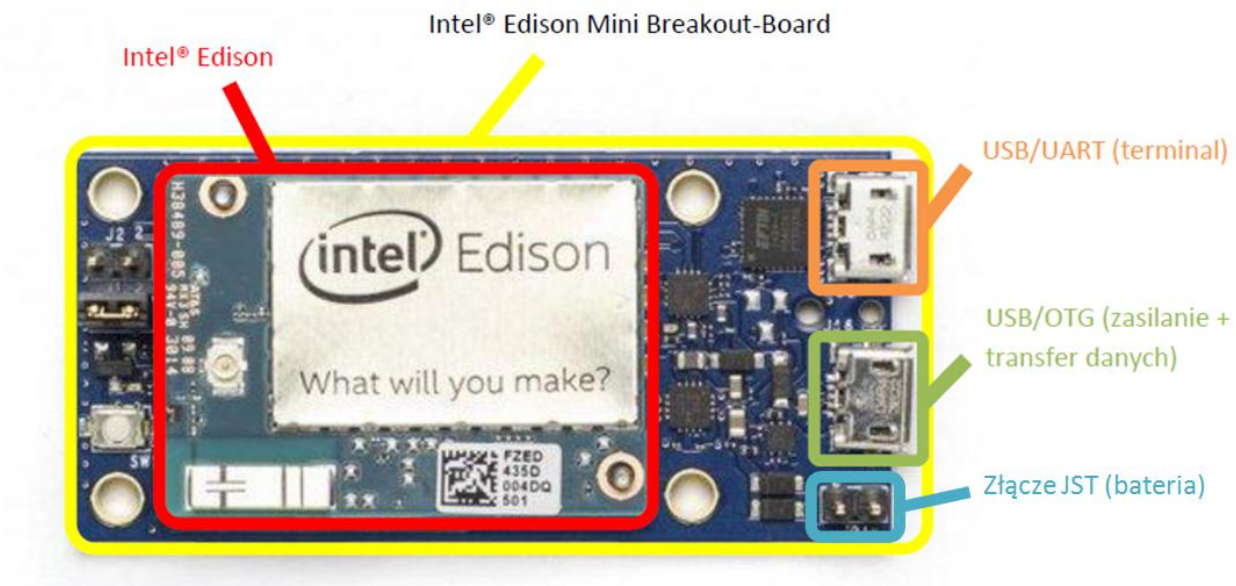
```
$ mosquitto_pub -h 192.168.43.250 -t ”Pressure” -p 8883 --cafile /certs/ca.crt -m “1002 hPa”
```

Komenda wysyłająca dane na chmurę:

```
$ mosquitto_pub -h 192.168.43.250 -t ”conf” -p 8883 --cafile /certs/ca.crt -m “send”
```

Złącza układu

Prezentowane są złącza mniejszej płytki rozwojowej ”Intel Edison Mini Breakout kit”



Opis montażu, uruchamiania, testowania układu

Montaż układu

Jako iż przy realizacji swojego projektu głównie korzystałem z płytki deweloperskiej: „Intel Edison Mini Breakout kit” montaż, nie stanowił większego problemu. Podłączenie modułu Edisona do płytki rozwojowej przeprowadziłem zgodnie z instrukcją producenta: https://software.intel.com/en-us/node/628223.

Uruchamianie układu

Uruchamianie oprócz pierwszego, który wymagał zainstalowania odpowiednich sterowników na komputerze-hoście oraz programów typu PuTTy i winSCP w celu komunikacji z Edisonem, również nie stanowiło większego problemu, gdyż do uruchomienia modułu, wystarczyło podpiąć zielone złącze (z obrazka Złącza układu) do zasilania po czym Edison, wraz z zainstalowanym na nim systemem operacyjnym automatycznie się uruchamiał.

Testowanie układu

Testowanie przeprowadziłem adekwatnie z celem projektu, czyli odbieraniem danych przesyłanych w protokole MQTT w lokalnej sieci Wi-Fi. Testy przeprowadziłem w taki sposób, że zainstalowałem testowego klienta (do pobrania tutaj: https://mosquitto.org/download/) na dwóch innych komputerach w lokalnej sieci (laptop i komputer PC) i testowałem prawidłowe odbieranie komunikatów przez Edisona jak i wysyłanie danych na chmurę (konto: edison.mqtt.data@gmail.com). Przetestowana została również certyfikacja SSL, bez odpowiednich plików certyfikujących na zewnętrznych komputerach, komunikaty do Edisona nie mogły być wysyłane. Działanie przetestowałem również w trzech różnych sieciach:

1. Sieć domowa (router Wi-Fi)
2. Sieć uczelniana (eduroam)
3. Sieć Hot-Spot, udostępniana przez mój telefon

We wszystkich działanie systemu jak i jego rezultat było takie same – prawidłowe. Jedyne co wymagało zmiany to ustawienie adresu IP Edisona – co wymuszało utworzenie nowych plików certyfikujących, dla każdej nowej sieci.

Wnioski z przebiegu pracy

Montaż i uruchamianie układu jak zostało wspomniane w poprzednim rozdziale, nie przysporzyły mi wiele problemów. Jednakże – co było głównym problemem projektu – część programistyczna/konfiguracyjna już nie należała do łatwych zadań. Po pierwsze musiałem zapoznać się z systemem jakim jest Linux Yocto 3.5 – wcześniej niestety nie miałem większego kontaktu z żadnym z Linuxów, dlatego sporo czasu musiałem poświecić na zapoznanie się z samym systemem, jego organizacją, sposobem obsługi, jak i stylem programowania pod niego. Co ważne – sama konfiguracja systemu, taka jak uruchamianie na nim usług, przyznawanie praw do plików, czy ustawianie wspomnianego adresu IP stanowiło niełatwe wyzwanie, gdyż większość poradników w sieci opiera się o systemy rodzin Debian, Ubuntu itd. Dla Yocto niewiele jest poradników w sieci, co prawda dużo rzeczy da się zrealizować w podobny sposób jak w innych dystrybucjach, jednakże często występowały jakieś drobne różnice, na których uporanie się z nim także trzeba było poświecić sporo czasu. Wiele taki szczegółów opisanych jest w udostępnianej przez mnie dokumentacji własnej na GitLabie: Dokumentacja/Moja_dokumentacja. Następnym problemem był wybór dystrybucji brokera, ja zdecydowałem się na Mosquitto firmy Eclipse, gdyż jest to wersja darmowa oraz open-sourcowa oraz z pośród wszystkich wersji najczęściej stosowana (co przekładało się na liczbę poradników w sieci). Jest ona również cały czas wspierana i co więcej rozwijana przez studio Eclipse. Kolejnym problemem był wybór języka programowania, w jakim zrealizuje potrzebne programy obsługujące brokera MQTT. Zdecydowałem się na Pythona – po pierwsze, jest to język który bardzo dobrze współpracuje z Linuxem (jednakże na Yocto jego interpreter wymagał aktualizacji do wersji 2.7.9). Po drugie, ponownie firma Eclipse udostępnia szeroką bibliotekę paho-MQTT, która jest dostępna w Pythonie, a której możliwości są naprawdę spore, jako jedna z nielicznych umożliwia wykorzystanie certyfikacji TLS/SSL w swoich programach. Pod koniec realizacji projektu, po konsultacjach z Prowadzącym, wysnuł się wniosek, aby dane przechowywane lokalnie na Edisonie (do momentu wysłania na chmurę) były przechowywane na zewnętrznej pamięci FRAM. Podyktowane to jest tym, że Edison, jako pamięć wewnętrzną wykorzystuje pamięć RAM typu Flash. Pamięć FRAM w porównani do pamięć RAM cieszy się o wiele większą przeżywalnością, którą szacuje się na ok. 100 trylionów odczytów/zapisów. Jednakże dokonałem szczegółowej analizy na tym polu – jak organizowana jest pamięć wewnętrzna Edisona. Otóż jest ona wykonana w technologii ”Managed NAND (eMMC) flash” z wykorzystaniem wear-levelingu oraz jej pojemność wynosi aż 4GB. Po pierwsze – manager-NAND – gwarantuje, że pamięć ta jest pamięcią nieulotną (czego wymaga chociażby przechowywanie całego systemu operacyjnego w tej pamięci). Po drugie – wear-leveling – to oprogramowanie w jakie wyposażona jest ta pamięć – służy do tego, że wszystkie komórki w pamięci są wykorzystywane po równo, co ma przełożyć się na ogólną żywotność pamięci. Producent dostarcza taki oto wzór, no oszacowanie żywotności pamięci wykorzystującej wear-leveling (oficjalny dokument producenta na moim GitLabie: Dokumentacja/Moja_dokumentacja/NAND_Wear-leveling.pdf)

The expected NAND Flash lifetime can be calculated as follows:

$$\text{Expected lifetime} = \frac{\text{Size of NAND flash} \times \textit{n}\text{umber of erase cycles} \times \text{FAT overhead}}{\textit{b}\text{ytes written per day}}$$

Przy czym producent Edisona zapewnia, ze pojedyncza komórka tej pamięci ma żywotność 100.000 odczytów/zapisów oraz pojemność pamięci wynosi 4GB. Współczynnik FAT overhead wynosi zazwyczaj 0,7. Zakładając że na dzień plik zapisywał by 1MB (co przekłada się na ok. 150 000 słów w pliku tekstowym – co

jest raczej zawyżone) oraz że tylko połowa pamięci była by zapisywana/odczytywana na nowo, łączny przewidywany czas żywotności dysku Edisona wynosi:

$$Expected\ lifetime = \frac{2\,000\,000\,000 \times 100\,000 \times 0.7}{1\,000\,000} = 140\,000\,000\ dni = 383\,561\ lat$$

A to głównie za sprawą pojemności pamięci Edisona, która wynosi aż 4GB, co w połączeniu z wykorzystaniem technologii wear-levelingu, daje na prawdę bardzo dobry rezultat.

Wnioski końcowe

Realizacja tego projektu pozwoliła mi zaznajomić się z elektroniką nie tylko na poziomie teorii, schematów i wykresów, ale również pod względem podejścia praktycznego do niej. Mój projekt – można powiedzieć – był bardziej programistyczny, jednakże kontakt z elektroniką także był odczuwalny. Przy realizacji tego projektu dowiedziałem się czym jest protokół komunikacyjny MQTT oraz jak lekki jest on w użyciu, a za razem jak wiele umożliwia. Dzięki temu projektowi zaznajomiłem się także bardziej z systemem operacyjnym jakim jest Linux, co skłoniło mnie nawet do przerzucenia się właśnie na ten system w życiu codziennym, na osobistym komputerze PC.

Literatura

[1] https://www.intel.com/content/dam/support/us/en/documents/edison/sb/edison-module-hardware-guide.pdf
[2] https://www.intel.com/content/dam/support/us/en/documents/edison/sb/edison_pb_331179002.pdf
[3] https://www.yoctoproject.org/docs/1.8/yocto-project-qs/yocto-project-qs.html
[4] http://www.steves-internet-guide.com/mosquitto-tls/
[5] https://github.com/prasmussen/gdrive