Android Things oraz Raspberry Pi 3 (1)

Wprowadzenie do systemu i pierwszy projekt

Artykuł ten jest pierwszym odcinkiem serii poświęconej systemowi Android Things i płytce Raspberry Pi. Pierwsza część cyklu przybliża Czytelnikowi krótką historię systemu a także zawiera opis przygotowania środowiska i pierwszej prostej aplikacji. W kolejnych częściach przedstawiony zostanie opis sterowania pracą interfejsu I²C, obsługi wyświetlacza OLED SSD1306 z wykorzystaniem gotowych sterowników sprzętu oraz projekt z zastosowaniem kamery i funkcji przetwarzania obrazu.

Trwająca od kilku lat rewolucja Internet of Things nabiera tempa, jednak brak jednolitych standardów powoduje, że rynek ten jest wciąż mocno zdefragmentowany, a każdy z producentów sprzętu i oprogramowania próbuje „zagarnąć" jak największy kawałek tortu dla siebie. Po swoich nie do końca udanych początkach z systemami Android@Home oraz Brillo, do działania ponownie wkracza jeden z największych graczy na rynku – Google. Czy połączenie zmodyfikowanego na potrzeby rynku IoT systemu Android wraz z najbardziej popularnymi platformami sprzętowymi może się nie udać? Sprawdźmy to!

Android Things – nowy/stary system operacyjny dla urządzeń IoT

Obserwując od kilku lat rosnący rynek urządzeń IoT można było odnieść mylne wrażenie, że firma Google pozostaje w cieniu rozpoczynającej się rywalizacji, a wszystkie dotychczasowe działania firmy skupione są wyłącznie na segmencie urządzeń mobilnych, gdzie system Android już od kilku lat wiedzie niepodważalny prym. W tym czasie organizacje/projekty takie jak IoTivity [1] czy AllJoyn [2] zaczynają zrzeszać pod banderą Linux Foundation największych graczy na rynku taki jak Intel, Samsung czy LG. Czy Google naprawdę „przespało" ten okres?

Nic bardziej mylącego! O pierwszych działaniach firmy Google na rynku urządzeń IoT (a uściślając – w obszarze systemów automatyki domowej) można było usłyszeć już w 2011 roku podczas konferencji Google I/O [3]. Zaprezentowany wówczas projekt Android@Home zakładał stworzenie kompletnego systemu automatyki domowej, której poszczególne urządzenia pracują pod kontrolą systemu operacyjnego Android (nazwanego wówczas dumnie „systemem operacyjnym dla Twojego domu"). Jednym z pierwszych oficjalnych produktów spod logo projektu Android@Home miała być inteligentna żarówka stworzona przy współpracy z firmą LightingScience. Niestety mimo wielu obiecujących zapowiedzi, idea projektu Android@Home upadła dość szybko – zapowiedziane produkty nie pojawiły się na rynku, firma LightingScience usunęła ze swojej strony materiały zapowiadające nowy projekt, a Google nie podjęło już w swych wystąpieniach tematu Android@Home.

Od czasu Android@Home mija kilka lat, gdy podczas kolejnej konferencji Google I/O w 2015 roku firma zapowiada projekt Brillo – bazujący na Androidzie system operacyjny dla niewielkich urządzeń IoT wyposażonych w 32–64 MB RAM. Projekt ten w początkowej fazie zostaje entuzjastycznie przyjęty, jednak brak w nazwie projektu – mogącego działać jak magnes – kluczowego słowa „Android" nie był przypadkowy. Główne środowisko deweloperskie projektu Brillo bazowało na języku C++ i nie przypominało ono deweloperom urządzeń mobilnych dobrze znanych „schematów" z pracy w środowisku Android. Być może właśnie to było przyczyną, że niewielu programistów skupionych wówczas wkoło projektu Android zdecydowało się na podjęcie eksperymentów z systemem Brillo. Z biegiem kolejnych miesięcy, projekt Brillo zaczął pokrywać się kurzem.

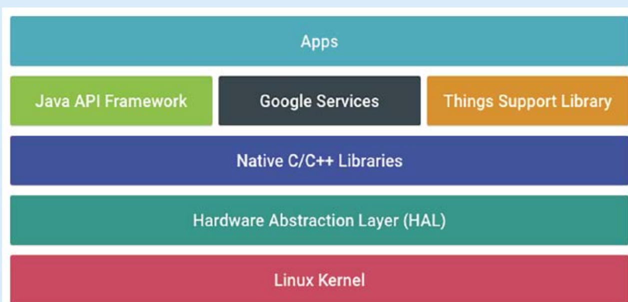
Sytuacja zmieniła się pod koniec roku 2016, kiedy to firma Google ogłosiła swój najnowszy projekt dla urządzeń IoT – system Android Things. „Nowy" projekt giganta z Mountain View jest właściwie kontynuacją systemu Brillo, w którym zdecydowano się uwzględnić większość krytycznych uwag pochodzących od deweloperów Brillo. Do najważniejszych zmian należy zaliczyć możliwość tworzenia aplikacji w języku Java z wykorzystaniem Android SDK, Google API oraz dobrze znanego programistom urządzeń mobilnych – pakietu Android Studio. Tym samym – jak zostanie to przedstawione w dalszej części niniejszego artykułu – przygotowanie aplikacji dla systemu Android Things niewiele różni się od tworzenia typowej aplikacji dla urządzeń mobilnych. Co więcej, firma Google przy współpracy z producentami najbardziej popularnych sprzętowych zestawów deweloperskich przygotowała gotowe obrazy dla takich platform jak: Intel Edison oraz Joule, NXP Pico i.MX7D, NXP i.MX6UL, NXP Argon i.MX6UL oraz Raspberry Pi 3. Tak więc wszystkie znaki na niebie i ziemi wskazują, że tym razem musi się udać.

Android Things – charakterystyka

Zanim przystąpimy do realizacji pierwszego projektu z wykorzystaniem Android Things, warto krótko scharakteryzować sam system i odpowiedzieć na pytanie, ile właściwie jest Androida w projekcie Android Things.

System Android Things jest zmodyfikowaną wersją standardowego projektu Android, dostosowaną do potrzeb typowych

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 95777, pass: 53wtjyf6



Rysunek 1. Warstwowy model programowy systemu Android Things (źródło: developer.android.com)

Tabela 1. Podsystemy Google API dostępne w Android Things (źródło: developer.android.com)

Wspierane podsystemy Google API	Niewspierane podsystemy Google API
Awareness	AdMob
Cast	Android Pay
Google Analytics for Firebase	Drive
Firebase Cloud Messaging (FCM)	Firebase App Indexing
Firebase Crash Reporting	Firebase Authentication
Firebase Realtime Database	Firebase Dynamic Links
Firebase Remote Config	Firebase Invites
Firebase Storage	Firebase Notifications
Fit	Play Games
Instance ID	Sign-In
Location	
Maps	
Nearby	
Places	
Mobile Vision	
SafetyNet	

systemów *embedded* (a więc w odróżnieniu od współczesnych telefonów – systemów pełniących jedną, ściśle określoną funkcję, jak np. sterowanie ogrzewaniem w systemie inteligentnego budynku). Ponieważ standardowe API systemu Android nie umożliwia programiście bezpośredniej obsługi najbardziej typowych dla systemów *embedded* komponentów sprzętowych (takich jak np. porty GPIO, PWM czy magistrale UART, SPI, I²C i I²S), interfejs programowy (API) systemu Android Things został rozbudowany o bibliotekę Things Support Library, umożliwiającą prostą obsługę „sprzętowego otoczenia" naszego projektu (**rysunek 1**).

Zgodnie z „charakterem pracy" urządzeń *embedded*, system operacyjny Android Things został zoptymalizowany pod kątem pracy z jedną aplikacją użytkownika, pełniącą funkcję głównej aktywności i uruchamianą tuż po starcie systemu. W urządzeniach z systemem Android Things wyświetlacz pełni funkcję opcjonalną i nie jest wymagany do pracy urządzenia. Dla wygody programistów chcących zbudować aplikację z graficznym interfejsem użytkownika, w systemie Android Things zdecydowano się pozostawić standardowy UI toolkit, umożliwiający budowanie funkcjonalnych interfejsów za pomocą graficznego edytora lub plików opisu XML.

W porównaniu do systemu standardowego systemu Android, usunięto główny pasek powiadomień i przycisków ekranowych (które przy jednej „aplikacji głównej" tracą rację bytu), a tym samym usunięto również mechanizm powiadomień. Projektant graficznego interfejsu użytkownika zyskuje tym samym do dyspozycji całą przestrzeń wyświetlanego obrazu. Interfejs graficzny może być obsługiwany za pomocą ekranu dotykowego i wirtualnej klawiatury lub, jeśli nasz zestaw deweloperski nie jest wyposażony w taki ekran – za pomocą dołączonej do portu USB standardowej myszy komputerowej.

Zmiana charakteru pracy systemu operacyjnego Android (z oprogramowania zarządzającego pracą wielofunkcyjnego telefonu na ściśle ukierunkowane urządzenie wbudowane) pozwoliła na „odchudzenie" systemu poprzez usunięcie aplikacji systemowych i wielu innych komponentów programowych. Tak więc, przygotowując aplikację dla systemu Android Things, warto pamiętać, że nie skorzystamy z takich standardowych dostawców treści (Content Providers) [4] jak: CalendarContract, ContactsContract, DocumentsContract, DownloadManager, MediaStore, Settings, Telephony, UserDictionary oraz VoicemailContract. System Android Things wspiera również skróconą listę komponentów z Google API – zbiorcze zestawienie umieszczono w **tabeli 1**.

Mimo znacznego odchudzenia Android Things, podczas pracy z systemem należy pamiętać, że wciąż mamy do czynienia ze standardowym ART (Android Runtime), które nie zawsze jest demonem wydajności. Niskopoziomowi programiści systemów *embedded* pracujący z językami C/C++ w dystrybucjach Linuksowych mogą odczuwać „niedosyt" z szybkości działania systemu (analizując go np. poprzez porównanie maksymalnej szybkości zmian stanów na wyprowadzeniach GPIO). Do kogo zatem jest adresowany Android Things?

Analizując dokumentację systemu, łatwo można odnieść wrażenie, że główną grupą docelową projektu są programiści obecnie skupieni wokół systemu Android i programowania urządzeń mobilnych. Właśnie dla tej grupy docelowej strony dokumentacji projektu zostały rozszerzone o krótki elementarz elektroniki cyfrowej i zwięzłe opisy wybranych magistral sprzętowych [5]. Poprzez udostępnienie dobrze znanego deweloperom środowiska programowania, Google chce przyciągnąć do swojego nowego projektu dużą rzeszę doświadczonych programistów Android, których w dynamicznej rozwijającej się branży IoT wciąż brakuje. Osobiście (jako elektronik, osoba programująca głównie w języku C i mająca na swoim koncie kilka aplikacji na system Android) również dostrzegam w projekcie Android Things duży potencjał. Łatwość tworzenia oprogramowania i GUI, przenośność, infrastruktura do przeprowadzania aktualizacji OTA oraz bardzo funkcjonalne i rozbudowane Google API (pozwalające tworzyć prototypy złożonych projektów w ciągu paru chwil) powoduje, że nie można przejść koło tego systemu obojętnie. Choć obecna wersja systemu – oznaczona jako „developer preview" – jest wolna i niestabilna, warto już teraz zapoznać się z możliwościami, jakie dziś oferuje projekt Android Things.

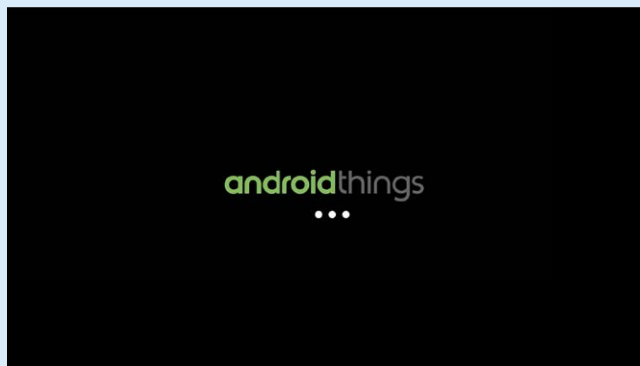
Przygotowanie platformy sprzętowej

W niniejszym artykule – do testów systemu Android Things – wybrano jedną z najbardziej popularnych płytek deweloperskich – Raspberry Pi 3. Przy współpracy z Raspberry Pi Foundation, firma Google przygotowała gotowe do pobrania obrazy systemu dostępne pod adresem ***https://goo.gl/eQLXKT***.

Procedura przygotowania karty pamięci SD z systemem Android Things jest analogiczna do instalacji innych systemów operacyjnych (Raspbian, Ubuntu, itp.) dla zestawu Raspberry Pi. W pierwszym kroku, w czytniku kart SD naszego komputera umieszczamy kartę pamięci o pojemności minimum 8 GB. Następnie po pobraniu i rozpakowaniu archiwum *androidthings_rpi3_devpreview_4_1.zip*, zawierającego obraz systemu Android Things, rozpoczynamy proces wgrywania obrazu na kartę SD. W systemie operacyjnym Linux, operację tę wykonamy za pomocą narzędzia *dd* [6] *dd bs=4M if=iot_rpi3.img of=/dev/sd<X>*, natomiast w środowisku Windows, wgranie obrazu systemu na kartę pamięci może zostać zrealizowane za pomocą narzędzia *Win32DiskImager* [7].

Po zakończonej sukcesem operacji instalacji obrazu przygotowaną kartę SD umieszczamy w slocie karty pamięci zestawu Raspberry Pi, natomiast do złącza HDMI podłączamy zewnętrzny monitor. Taki minimalny zestaw połączeń pozwoli nam upewnić się, że przygotowanie karty SD z systemem zostało zrealizowane

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 95777, pass: 53wtjyf6



Rysunek 2. Animacja z logo Android Things wyświetlana przy starcie systemu

w prawidłowy sposób. Po włączeniu zasilania na ekranie zostanie wyświetlono animacja startowa, jak na **rysunku 2**.

Należy podkreślić, że pierwsze uruchomienie Android Things jest dość czasochłonne (proces ten może trwać około 3 minut), a pomiędzy kolejnymi etapami startu systemu mogą występować momenty, kiedy ekran nie wyświetla żadnego obrazu. Ponieważ dioda aktywności systemu umieszczona na płytce Raspberry Pi nie jest obsługiwana, urządzenie może stwarzać wrażenie „zawieszonego" – mając jednak na uwadze, że Android Things to wciąż wersja „developer preview" – uzbrójmy się w chwilę cierpliwości.

Ponieważ w systemie nie została zainstalowana jeszcze aplikacja użytkownika, którą system mógłby uruchomić tuż po starcie, po zakończeniu procesu uruchamiania systemu na ekranie urządzenia zostanie wyświetlone logo systemu Android Things wraz z informacjami o aktualnym stanie połączeń sieciowych (które w omawianym przypadku nie zostały jeszcze ustanowione). Jak wspomniano w poprzednim podrozdziale – system Android Things został zoptymalizowany pod kątem uruchomienia wyłącznie jednej aplikacji użytkownika, więc nie został on wyposażony w żadne menu pozwalające na jakąkolwiek interakcję z systemem poprzez graficzny interfejs użytkownika. Aby rozpocząć pracę z systemem, niezbędne jest na tym etapie ustanowienie połączenia sieciowego. Jedną z najprostszych i najbardziej stabilnych metod jest wykorzystanie kabla Ethernet – tuż po podłączeniu zestawu Raspberry Pi do sieci lokalnej ekran startowy systemu poinformuje nas o ustanowionym połączeniu i przydzielonym adresie IP.

Istnieje również możliwość podłączenia zestawu Raspberry Pi do sieci Wi-Fi, jednak w przypadku systemu Android Things, zestawienie takiego połączenia jest bardziej pracochłonne. Ponieważ graficzny ekran startowy systemu uniemożliwia nam przeprowadzenie jakichkolwiek interakcji z systemem, do ustanowienia połączenie Wi-Fi niezbędne jest uprzednie zalogowanie się do konsoli systemu poprzez port UART, z wykorzystaniem sprzętowego konwertera *UART TTL<–>USB* oraz wybranego oprogramowania umożliwiającego obsługę portu szeregowego w konfiguracji *115200 8N1*. Schemat wymaganych połączeń dla zestawu Raspberry Pi 3 został przedstawiony na **rysunku 3**.

Po zalogowaniu się do konsoli systemu, użytkownik ma możliwość podłączenia do wybranej sieci Wi-Fi poprzez polecenie:

```
$ am startservice \
      -n com.google.wifisetup/.WifiSetupService \
      -a WifiSetupService.Connect \
      -e ssid <NAZWA SIECI> \
      -e passphrase <HASŁO>
```

Poprawność nawiązania połączenia może zostać zweryfikowana poleceniem:

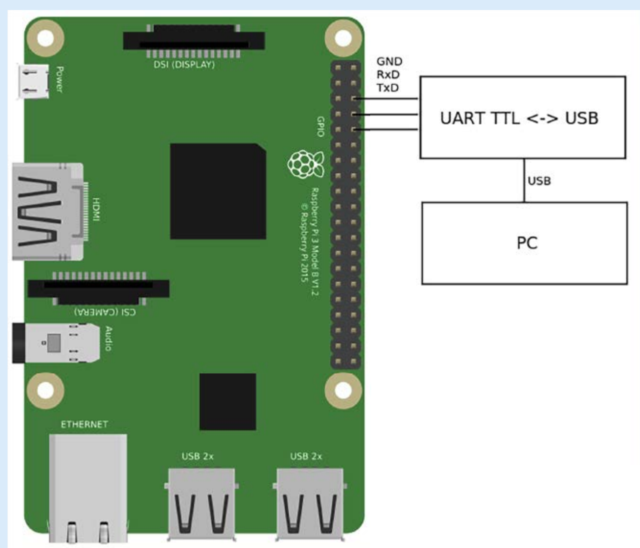
```
$ logcat -d | grep Wifi
...
V WifiWatcher: Network state changed to CONNECTED
V WifiWatcher: SSID changed: ...
I WifiConfigurator: Successfully connected to ...
```

Jeżeli konfiguracja sieci została przeprowadzona prawidłowo, na ekranie startowym zostanie wyświetlony przydzielony adres IP, a ustawienia sieci zostaną zapisane – urządzenie spróbuje zestawić to samo połączenie przy kolejnym starcie systemu. Aby skasować listę zapisanych sieci Wi-Fi, w terminalu systemu należy wydać polecenie:

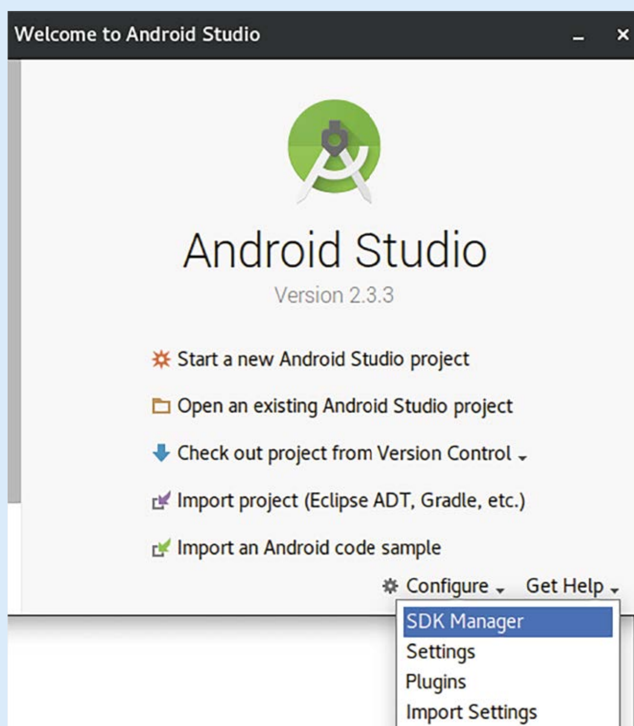
```
$ am startservice \
      -n com.google.wifisetup/.WifiSetupService \
      -a WifiSetupService.Reset
```

Przygotowanie środowiska programistycznego

Jak wspomniano we wstępnie, jedną z niewątpliwych zalet systemu Android Things jest możliwość pracy w wygodnym

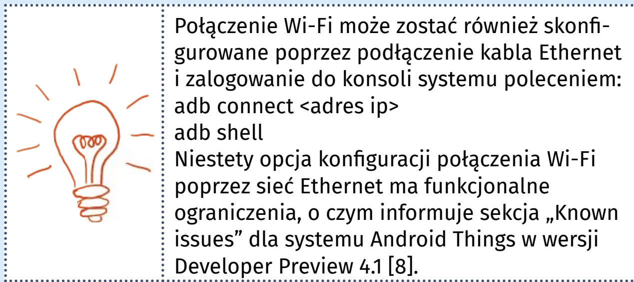


Rysunek 3. Podłączenie sprzętowego konwertera UART TTL↔USB do zestawu Raspberry Pi 3

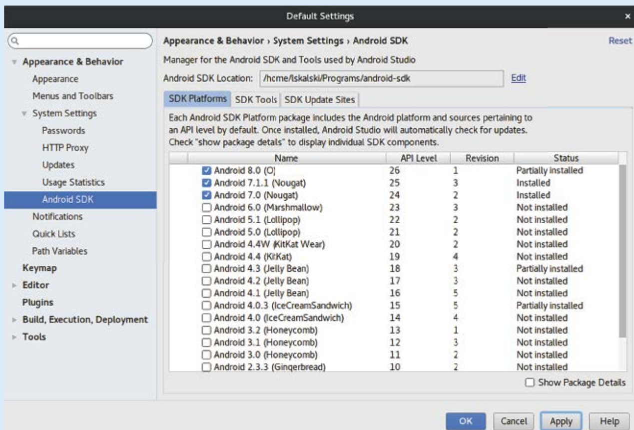


Rysunek 4. Uruchomienie menedżera „SDK Manager" przed utworzeniem pierwszego projektu

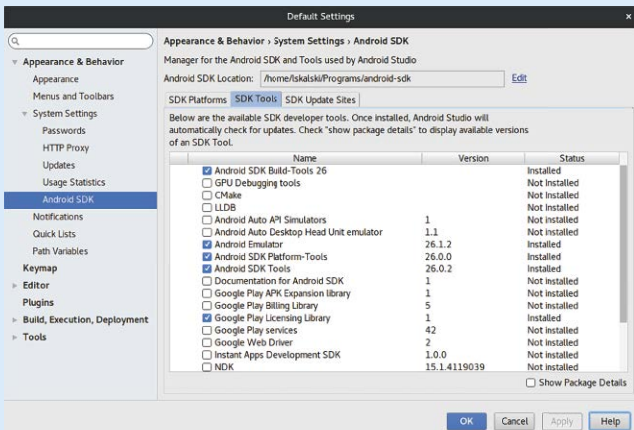
Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **95777**, pass: **53wtiyf6**



Połączenie Wi-Fi może zostać również skonfigurowane poprzez podłączenie kabla Ethernet i zalogowanie do konsoli systemu poleceniem:
adb connect <adres ip>
adb shell
Niestety opcja konfiguracji połączenia Wi-Fi poprzez sieć Ethernet ma funkcjonalne ograniczenia, o czym informuje sekcja „Known issues" dla systemu Android Things w wersji Developer Preview 4.1 [8].



Rysunek 5. Instalowanie „SDK Platforms" w wersji >= API 24



Rysunek 6. Instalacja „SDK Tools" w wersji >= 25.0.3

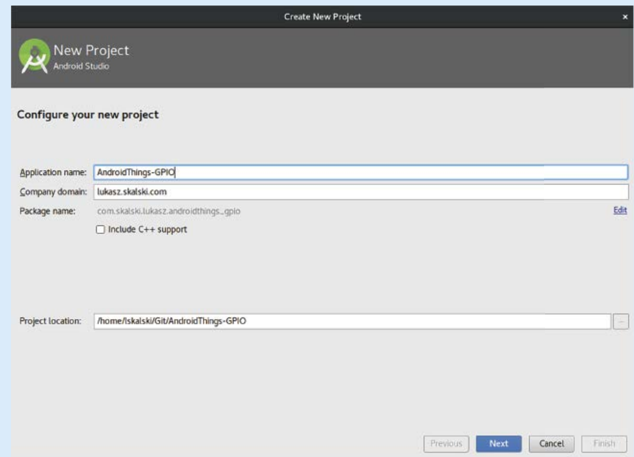
i funkcjonalnym środowisku Android Studio, będącym oficjalnym IDE dla systemu Android. Ostatnią stabilną wersję pakiet Android Studio (w chwili tworzenia artykułu jest to wersja oznaczona numerem 2.3.3) można pobrać pod adresem ***https://goo.gl/6qBMkm***.

Instalowanie środowiska w systemie Windows jest realizowane poprzez standardowy instalator, wymagający jedynie wskazania docelowego folderu instalacji. W systemie operacyjnym Linux „instalacja" ogranicza się do rozpakowania pobranego archiwum ZIP i uruchomienia skryptu *android-studio/bin/studio.sh*.

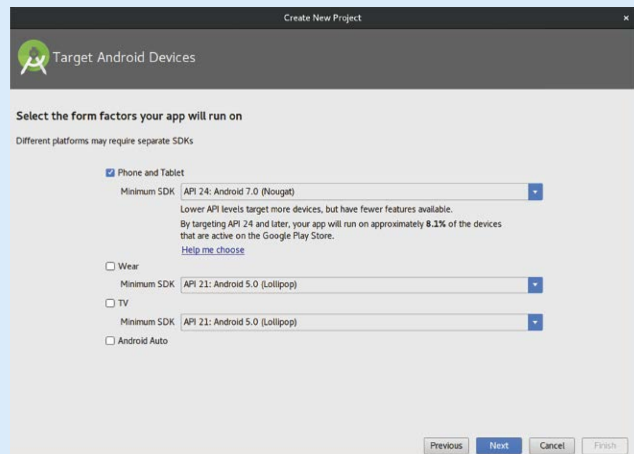
Przy pierwszym uruchomieniu środowiska Android Studio, niezbędne jest uruchomienie menedżera oprogramowania SDK Manager (**rysunek 4**) oraz pobranie najnowszych narzędzi SDK Tools w wersji 25.0.3 lub wyżej, oraz SDK Platforms dla Androida 7.0 (wersja API 24) – jak przedstawiono to na **rysunku 5** oraz **rysunku 6**.

Pierwszy projekt – sterowanie GPIO

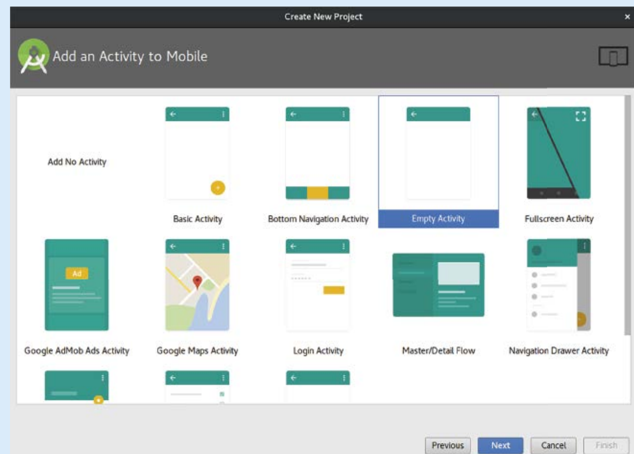
Zarówno platforma sprzętowa, jak i środowisko programistyczne jest już gotowe na przygotowanie pierwszego prostego projektu. Zgodnie z niepisaną zasadą, będzie to *Hello World* w wersji na systemy *embedded*, czyli prosty projekt obsługujący diody LED oraz jeden przycisk.



Rysunek 7. Kreator nowego projektu – etap 1/4



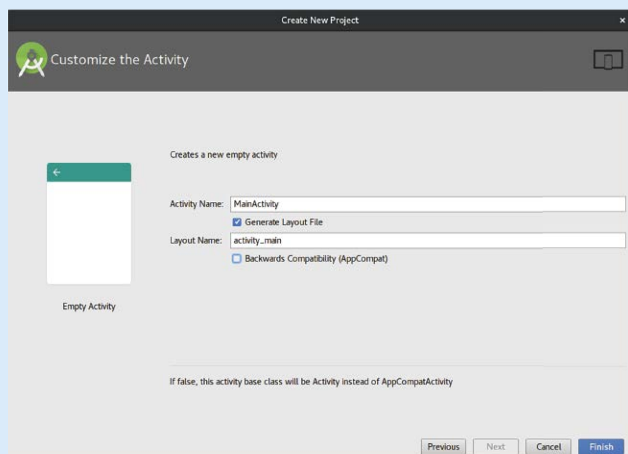
Rysunek 8. Kreator nowego projektu – etap 2/4



Rysunek 9. Kreator nowego projektu – etap 3/4

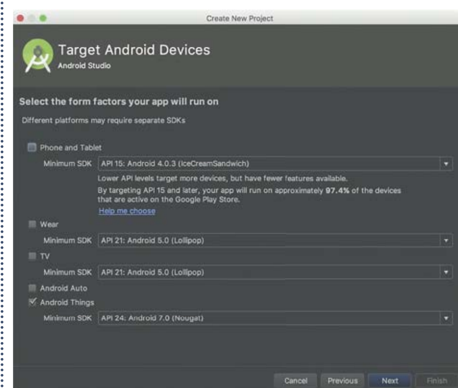
Po uruchomieniu Android Studio, wybieramy opcję *Start a new Android Studio project*, która przenosi nas do okna konfiguracji nowego projektu, jak na **rysunku 7**. W następnym kroku użytkownik zostanie poproszony o określenie typu projektu. Ponieważ oficjalne wsparcie dla platformy Android Things pojawi się w kolejnej wersji Android Studio (wersja 3.0 – patrz ramka poniżej), jako typ projektu wybieramy opcję *Phone and Tablet* ze wskazaniem minimalnej wersji API 24 (Android 7.0) – **rysunek 8**. Pomimo że nasza pierwsza aplikacja nie będzie wykorzystywała graficznego interfejsu użytkownika (przykłady z GUI zostaną przedstawione w kolejnych artykułach z cyklu), na kolejnym etapie konfiguracji wybieramy opcję *Empty Activity* – generator projektu utworzy wówczas domyślną aktywność aplikacji (*MainActivity.java*) oraz

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl, user: 95777, pass: 53wfjyf6**



Rysunek 10. Kreator nowego projektu – etap 4/4

W ramach opcji „Early access" firma Google udostępniła deweloperom najnowszą wersję pakietu Android Studio oznaczoną numerem 3.0. Wersja ta jest dostępna do pobrania pod adresem:
https://goo.gl/AjUn3b
Z perspektywy niniejszego artykułu, do najważniejszych zmian w wersji 3.0 należy zaliczyć możliwość bezpośredniego tworzenia projektów dla systemu Android Things, korzystając z wbudowanego kreatora projektu.



prosty układ (*layout/activity_main.xml*) wyświetlający na ekranie napis *Hello World* (**rysunek 9** oraz **rysunek 10**).

Na obecnym etapie konfiguracji projektu przygotowanie aplikacji jest tożsame z tworzeniem aplikacji dla urządzeń mobilnych. Pierwszym z wyróżników jest konieczność włączenia wsparcia dla biblioteki Things Support Library, niedostępnej w standardowym API. W tym celu, edytujemy plik *app/build.gradle* w sekcji *dependencies*:

```
dependencies {
    ...
    provided ,com.google.android.things:androidthings:0.4.1-devpreview'
}
```

oraz plik *AndroidManifest.xml*:

```
<application ...>
    <uses-library android:name="com.google.android.things"/>
    ...
</application>
```

Pozostając przy edycji pliku *AndroidManifest.xml*, dokonajmy jeszcze jednej zmiany, która wyróżnia tworzenie aplikacji dla systemu Android Things. Jak wspomniano, system Android Things został zaprojektowany do uruchomienia wyłącznie jednej aplikacji użytkownika, tuż po starcie systemu. W tym celu, poprzez wpisy *intent-filter* [9] należy określić, która z aktywności ma stanowić „punkt wejścia" aplikacji. Zmodyfikowany na potrzeby systemu Android Things plik *AndroidManifest.xml* został przedstawiony na **listingu 1**.

Listing 1. Plik AndroidManifest.xml zmodyfikowany na potrzeby systemu Android Things

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.skalski.lukasz.androidthings_gpio">
    <application
        android:label="@string/app_name">
        <uses-library android:name="com.google.android.things"/>
        <activity android:name=".MainActivity">
            <!-- Launch activity as default from Android Studio -->
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
            <!-- Launch activity automatically on boot -->
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.IOT_LAUNCHER"/>
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Listing 2. Szablon klasy MainActivity wygenerowany przez kreator projektu

```
package com.skalski.lukasz.androidthings_gpio;
import android.app.Activity;
import android.os.Bundle;
public class MainActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Na obecnym etapie edycji projektu uzyskano gotowy szkielet aplikacji dla systemu Android Things. Aby sprawdzić poprawność wprowadzonych zmian, zbudujmy projekt poprzez wybranie opcji *Build* → *Make Project* (lub wybierając skrót klawiszowy *Ctrl+F9*).

Zanim przystąpimy do wykonania połączeń sprzętowych (podłączenia diod LED i przycisku) oraz implementacji obsługi programowej, spróbujmy uruchomić szkielet naszej aplikacji na platformie docelowej. Uruchomienie zbudowanej uprzednio aplikacji odbywa się poprzez wybranie opcji *„Run* → *Run 'app'"* (lub skrótu klawiszowego *Shift+F10*). Użytkownik zostanie poproszony o wybranie platformy docelowej, jak przedstawiono to na **rysunku 11**. Ponieważ nie zestawiliśmy dotychczas połączenia z platformą Raspberry Pi, lista dostępnych urządzeń jest pusta. Aby nawiązać połączenie z docelową platformą deweloperską, za pomocą terminalu systemu operacyjnego wydajemy polecenie:

```
[user@archlinux ~]$ adb connect <adres ip>:5555
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
connected to 192.168.0.129:5555
```

Aby upewnić się, że połączenie zostało zestawione prawidłowo, wykorzystując ponownie program *adb* [10], możemy wyświetlić listę aktualnie podłączonych urządzeń:

```
[user@archlinux ~]$ adb devices
List of devices attached
192.168.0.129:5555    device
```

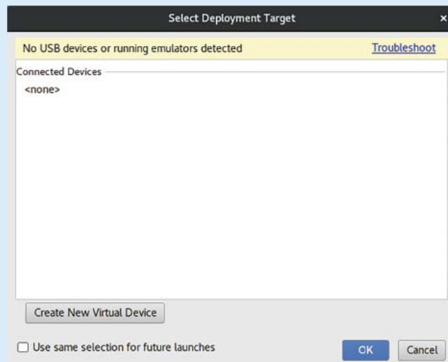
Tym samym lista urządzeń w programie Android Studio, na których możemy uruchomić szkielet naszej aplikacji, została poszerzona o nowy wpis (**rysunek 12**).

Wybierając docelową platformę, rozpoczynamy proces instalacji aplikacji. Ponieważ jest to proces dość czasochłonny,

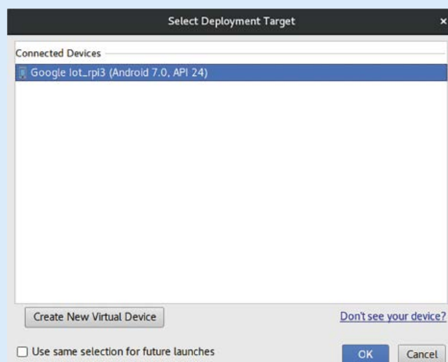
Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **95777**, pass: **53wtjyf6**

Poprzednie części kursu i dodatkowe materiały dostępne są na na FTP: ftp://ep.com.pl, user: 95777, pass: 53wjyf6

w oczekiwaniu na wyświetlenie napisu *Hello World*, zrealizujmy na płytce prototypowej połączenia diod LED i przycisku. Zadaniem przykładowej aplikacji będzie wyświetlenie prostych efektów świetlnych na dwóch diodach LED, przełączanych za pomocą pojedynczego przycisku. Schemat połączeń przedstawiono na **rysunku 13**.



Rysunek 11. Okno wyboru platformy docelowej przed nawiązaniem połączenia z Raspberry Pi



Rysunek 12. Okno wyboru platformy docelowej po nawiązaniu połączenia za pomocą programu ADB

Poprawne uruchomienie szkieletu naszej aplikacji oraz wykonanie wszystkich połączeń sprzętowych umożliwia nam przejście do finalnego kroku tej części artykułu – dopisania właściwej obsługi sprzętowej w pliku *MainActivity.java*, którego wstępna postać, wygenerowana przez kreator projektu, została przedstawiona na **listingu 2**.

Każda aplikacja przygotowana na platformę Android ma własny „cykl życia", podczas którego może znajdować się ona w jednym z określonych stanów [11] (m.in. aplikacja jest uruchamiana, zatrzymana, przerwana lub zakończona). Programista aplikacji nie ma wpływu na stan, w jakim obecnie znajduje się aplikacja, jednak może zapewnić prawidłową programową obsługę każdego z tych stanów. Obsługa ta jest realizowana poprzez przedefiniowanie kilku kluczowych metod (takich jak m.in. *onCreate()*, *onDestroy()*, *onResume()*, *onPause()*) w klasie dziedziczącej po klasie *android.app.Activity*. Cykl życia aplikacji Android wymaga, aby została wywołana metoda *onCreate()*, którą każda klasa pochodna od *Activity* (w tym *MainActivity* z omawianego przykładu) nadpisuje celem wykonania własnych operacji inicjalizujących. Ponieważ w systemie Android Things mamy do czynienia wyłącznie z jedną aplikacją użytkownika, która nie może być przeniesiona do „tła" celem uruchomienia innej aplikacji, nasz program znajduje się w jednym z dwóch stanów: uruchomienia lub zatrzymania. Tym samym rolą programisty jest nadpisanie metody *onCreate()* – odpowiadającej za inicjalizację aplikacji oraz metody *onDestroy()*, która umożliwi poprawne jej zamknięcie. Zaktualizowany kod klasy *MainActivity* został przedstawiony na **listingu 3**.

Listing 3. Szablon klasy MainActivity wygenerowany przez kreator projektu

```
public class MainActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
      /* INICJALIZACJA PROGRAMU */
    }
    @Override
    protected void onDestroy()
    {
        super.onDestroy();
      /* ZAKONCZENIE PROGRAMU */
    }
}
```

Tabela 2. Wybrane metody klasy PeripheralManagerService [12]

Zwracany typ	Funkcja
Gpio	openGpio(String name)
I2cDevice	openI2cDevice(String name, int address)
Pwm	openPwm(String name)
SpiDevice	openSpiDevice(String name)
UartDevice	openUartDevice(String name)

Tabela 3. Wybrane metody klasy Gpio [13]

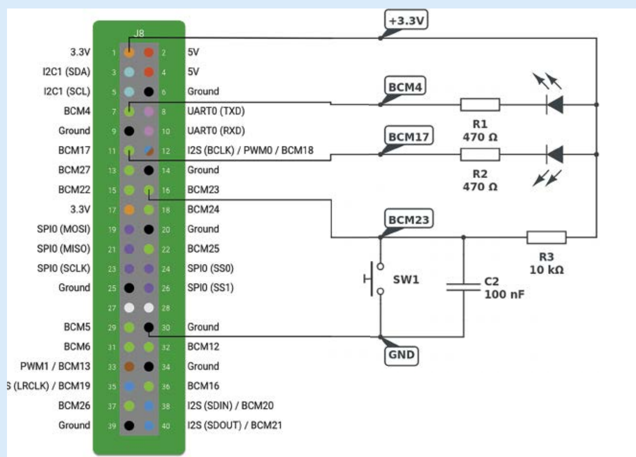
Funkcja	Zadanie
abstract void setDirection (int direction)	Ustaw kierunek pracy portu GPIO (wejście/wyjście)
abstract boolean getValue()	Odczytaj stan logiczny wyprowadzenia GPIO (tylko dla wyprowadzeń typu „wejście")
abstract void setValue (boolean value)	Ustaw stan logiczny wyprowadzenia GPIO (tylko dla wyprowadzeń „wyjście")
abstract void setEdgeTriggerType (int edgeTriggerType)	Określ rodzaj zbocza generującego zdarzenie
final void registerGpioCallback (GpioCallback callback)	Zarejestruj funkcję zwrotną obsługi zdarzenia
abstract void unregisterGpioCallback (GpioCallback callback)	Odrejestruj funkcję zwrotną obsługi zdarzenia

Listing 4. Konfiguracja wyprowadzeń BCM4 oraz BCM17

```
PeripheralManagerService service = new PeripheralManagerService();
Gpio led_1 = service.openGpio(„BCM4”);
Gpio led_2 = service.openGpio(„BCM17”);
led_1.setDirection(Gpio.DIRECTION_OUT_INITIALLY_LOW);
led_2.setDirection(Gpio.DIRECTION_OUT_INITIALLY_HIGH);
```

Listing 5. Konfiguracja wyprowadzenia BCM23, pełniącego funkcję wejścia

```
Gpio button = service.openGpio(„BCM23”);
button.setDirection(Gpio.DIRECTION_IN);
```



Rysunek 13. Schemat podłączenia zewnętrznych komponentów sprzętowych

```
Listing 6. Konfiguracja zbocza sygnału i funkcji obsługi zdarzenia
button.setEdgeTriggerType(Gpio.EDGE_FALLING);
button.registerGpioCallback(ButtonCallback);
```

```
Listing 7. Funkcja obsługi przycisku
private GpioCallback ButtonCallback = new GpioCallback()
{
    @Override
    public boolean onGpioEdge(Gpio gpio)
    {
        Log.i(TAG, „Button pressed”);
        led_mode = !led_mode;
        try
        {
            if (led_mode)
            {
                led_1.setValue(false);
                led_2.setValue(false);
            } else
            {
                led_1.setValue(true);
                led_2.setValue(false);
            }
        }
        catch (IOException e)
        {
            Log.e(TAG, „Error on PeripheralIO API”, e);
        }
            /* return true to keep callback active */
        return true;
    }
};
```

W systemie *Android Things*, dostęp do określonych zasobów sprzętowych (takich jak: GPIO, I²C, I²S, SPI, UART czy PWM) realizowany jest poprzez obiekt *PeripheralManagerService* [12] z biblioteki Things Support Library oraz jedną z szeregu metod przedstawionych w **tabeli 2**.

Konfiguracja portów GPIO jest natomiast realizowana poprzez zbiór metody klasy GPIO [13] – **tabela 3**. Inicjalizacja wyprowadzeń *BCM4* oraz *BCM17* jako portów wyjściowych (do których zostały podłączone diody LED) została przedstawiona na **listingu 4**.

W analogiczny sposób konfigurujemy przycisk podłączony do wyprowadzenia *BCM23*, pełniącego funkcję wejścia – **listing 5**.

Konfiguracja wejścia wymaga również określenia zbocza sygnału, które wyzwoli funkcję obsługi przycisku, wskazaną poprzez *registerGpioCallback()* – **listing 6**.

W ramach aktywności *MainActivity* zaimplementujmy więc również funkcję zwrotną *ButtonCallback*, odpowiedzialną za obsługę przycisku i zmianę wyświetlanego efektu – **listing 7**.

Do uzyskania finalnego efektu pozostało jedynie poprawne zamknięcie aplikacji poprzez zwolnienie zasobów w metodzie *onDestroy()* oraz implementacja prostej funkcji (z wykorzystaniem interfejsu *Runnable* [14]), która z zadanym interwałem czasowym zmieni stan logiczny na wyprowadzeniach *BCM4* oraz *BCM17*. Kompletny kod klasy *MainActivity* został przedstawiony na **listingu 8**.

Łukasz Skalski

Linki zewnętrzne:

1. https://goo.gl/WzJu5p
2. https://goo.gl/jbSDuS
3. https://goo.gl/sDYK7V
4. https://goo.gl/Es3zWk
5. https://goo.gl/QUGeCX
6. https://goo.gl/mK9Dgu
7. https://goo.gl/yLZ8nh
8. https://goo.gl/eUYxqd
9. https://goo.gl/UcL8GN
10. https://goo.gl/EkPR28
11. https://goo.gl/Q7d3YT
12. https://goo.gl/HgLNyA
13. https://goo.gl/mX8GPR
14. https://goo.gl/6XmXVq


```
Listing 8. Kompletny kod klasy MainActivity
package com.skalski.lukasz.androidthings_gpio;
import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.util.Log;
import java.io.IOException;
import com.google.android.things.pio.Gpio;
import com.google.android.things.pio.GpioCallback;
import com.google.android.things.pio.PeripheralManagerService;
public class MainActivity extends Activity
{
    private static final String TAG = „MainActivity”;
    private static final int INTERVAL = 200;
    private Gpio button;
    private Gpio led_1;
    private Gpio led_2;
    private static final String BUTTON_NAME = „BCM23”;
    private static final String LED_1_NAME = „BCM4”;
    private static final String LED_2_NAME = „BCM17”;
    private Handler led_handler = new Handler();
    private boolean led_mode = false;
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        PeripheralManagerService service = new PeripheralManag-
erService();
        try
        {
            button = service.openGpio(BUTTON_NAME);
            button.setDirection(Gpio.DIRECTION_IN);
            button.setEdgeTriggerType(Gpio.EDGE_FALLING);
            button.registerGpioCallback(ButtonCallback);
            led_1 = service.openGpio(LED_1_NAME);
            led_2 = service.openGpio(LED_2_NAME);
            led_1.setDirection(Gpio.DIRECTION_OUT_INITIALLY_
LOW);
            led_2.setDirection(Gpio.DIRECTION_OUT_INITIALLY_
HIGH);
            led_handler.post(LedRunnable);
        }
            catch (IOException e)
        {
            Log.e(TAG, „PeripheralIO API ERROR”, e);
        }
    }

    private Runnable LedRunnable = new Runnable()
    {
        @Override
        public void run()
        {
            if (led_1 == null || led_2 == null) return;
            try
            {
                /* toggle the LED state */
                led_1.setValue(!led_1.getValue());
                led_2.setValue(!led_2.getValue());
                /* schedule another event after delay */
                led_handler.postDelayed(LedRunnable, INTERVAL);
```

```
Listing 8. cd.
            }
            catch (IOException e)
            {
                Log.e(TAG, „PeripheralIO API ERROR”, e);
            }
        }
    };

    private GpioCallback ButtonCallback = new GpioCallback()
    {
        @Override
        public boolean onGpioEdge(Gpio gpio)
        {
            Log.i(TAG, „Button pressed”);
            led_mode = !led_mode;
            try
            {
                if (led_mode)
                {
                    led_1.setValue(false);
                    led_2.setValue(false);
                } else
                {
                    led_1.setValue(true);
                    led_2.setValue(false);
                }
            }
            catch (IOException e)
            {
                Log.e(TAG, „Error on PeripheralIO API”, e);
            }
            /* return true to keep callback active */
            return true;
        }
    };
    @Override
    protected void onDestroy()
    {
        super.onDestroy();
        /* remove handler events on close */
        led_handler.removeCallbacks(LedRunnable);
        try
        {
            if (led_1 != null)
                led_1.close();
            if (led_2 != null)
                led_2.close();
            if (button != null)
            {
                button.unregisterGpioCallback(ButtonCallback);
                button.close();
            }
        }
        catch (IOException e)
        {
            Log.e(TAG, „Error on PeripheralIO API”, e);
        }
    }
}
```


Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 95777, pass: 53wtjyf6