Android Things oraz Raspberry Pi 3 (3)

Obrazy spersonalizowane. Aktualizacja oprogramowania przez Android Things Console

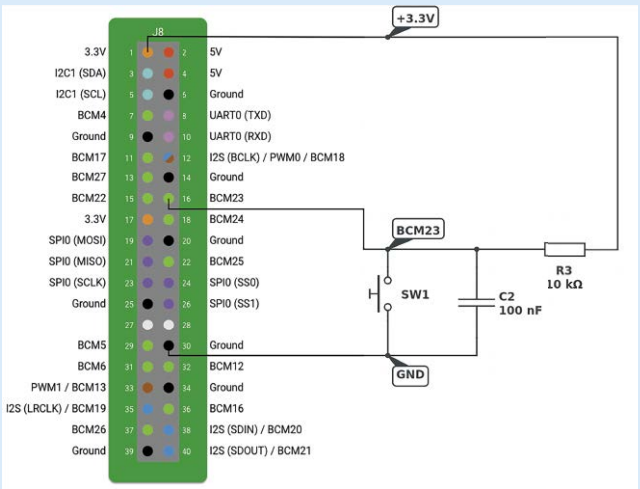
System Android Things jest dopiero wczesną wersją deweloperską, jednak już w obecnej fazie rozwoju dostarcza wielu funkcjonalności, które mogą pozytywnie wpłynąć na sukces tego projektu. Ciekawym i wartym przybliżenia rozwiązaniem jest platforma Android Things Console, która umożliwia niezwykle łatwe i szybkie budowanie spersonalizowanych obrazów oraz przeprowadzanie zdalnych aktualizacji OTA (Over The Air).

W tej części cyklu poświęconej systemowi Android Things oraz płytce Raspberry Pi, zbudujemy spersonalizowany obraz systemu – z własnym logo startowym oraz domyślnie zainstalowaną aplikacją użytkownika, której zadaniem będzie obsługa modułu kamery. Następnie – za pomocą narzędzia Android Things Console – przeprowadzona zostanie zdalna aktualizacja aplikacji. Zaczynajmy!

Obsługa modułu kamery Raspberry Pi

Zanim przystąpimy do budowy obrazu systemu z preinstalowaną aplikacją użytkownika, niezbędne jest uprzednie przygotowanie samej aplikacji. Aby nadać temu artykułowi pewien scenariusz oraz większy walor dydaktyczny, załóżmy, że postawionym przed nami zadaniem jest budowa systemu „inteligentnego dzwonka" dla jednego z naszych klientów. W systemie tym – po wciśnięciu przycisku dzwonka – za pomocą dołączonego modułu kamery [1] wykonujemy zdjęcia odwiedzających nas gości. Jednym z wymagań stawianych przez naszego „fikcyjnego klienta" jest to, aby poprzez dołączony do urządzenia wyświetlacz, prezentowana była lista dziesięciu ostatnich zdjęć, wraz z datą i godziną ich wykonania. Po zapełnieniu listy (wykonaniu serii 10 fotografii) lista powinna być czyszczona, a cały proces zostanie powtórzony od początku. Mając tak przedstawiony zarys wymagań, przystąpmy do realizacji zdania.

Proces przygotowania nowej aplikacji rozpoczynamy od podłączenia modułu kamery, wykonania połączeń sprzętowych dla przycisku (**rysunek 1**) oraz utworzenia nowego projektu, zgodnie z wytycznymi dla systemu *Android Things*, co opisano w pierwszym artykule opublikowanym w Elektronice Praktycznej 11/2017.



Rysunek 1. Schemat połączeń sprzętowych

Tworzenie kodu obsługi modułu kamery rozpoczynamy od zdefiniowania uprawnień, wymaganych do uzyskania dostępu do podłączonego sprzętu i sieci Internet. W tym celu w pliku *AndroidManifest.xml*, definiujemy poniższe uprawnienie:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.INTERNET" />
```

Uprawnienie dające nam dostęp do zasobów kamery jest zaliczane do kategorii uprawnień „niebezpiecznych", stąd przy pierwszej instalacji aplikacji niezbędne jest ponowne uruchomienie urządzenia (w odróżnieniu od uprawnień „bezpiecznych", które są przyznawane aplikacji na etapie jej instalacji).

Operacje związane z dostępem i obsługą urządzeń peryferyjnych są operacjami blokującymi, dlatego też, aby uniknąć problemów z brakiem responsywności interfejsu użytkownika, bardziej złożone i czasochłonne działania „na sprzęcie", powinny być wykonane w osobnym wątku. W tym celu, w głównej klasie aplikacji (*MainActivity.java*), utwórzmy nowy wątek tła oraz powiązany z nim *Handler* [2], jak pokazano na **listingu 1**.

Listing 1. Utworzenie wątku na potrzeby obsługi modułu kamery

```
public class MainActivity extends Activity
{
    /.../
    private Handler cameraHandler;
    private HandlerThread cameraThread;

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /.../

        /* create new handler and associated thread for camera
*/
        cameraThread = new HandlerThread(„CameraBackground");
        cameraThread.start();
        cameraHandler = new Handler(cameraThread.getLooper());

        /.../
    }

    /*
     * onDestroy
     */
    @Override
    protected void onDestroy() {
        super.onDestroy();

        cameraThread.quitSafely();
    }
}
```

Ponieważ kod obsługi modułu kamery jest generyczny i może zostać ponownie wykorzystany do budowy innych aplikacji, zostanie on umieszczony w ramach osobnej klasy *RPiCamera* (plik *RPiCamera.java*). Do zadań metody *initializeCamera()* (**listing 2**), utworzonej w ramach klasy *RPiCamera*, należeć będzie:

A. Wykorzystanie serwisu systemowego *CameraManager* [3] oraz funkcji *getCameraIdList()* do pobrania listy identyfikatorów wszystkich kamer dostępnych w systemie.
B. Utworzenie instancji *ImageReader*'a [4], której zadaniem będzie odczyt danych z modułu kamery oraz przetworzenie ich na obraz w formacie JPEG (o wielkości 640×480 pikseli). Przetwarzanie będzie realizowane w sposób asynchroniczny, a o jego zakończeniu i dostępności gotowego obrazu do dalszego przetwarzania, użytkownik będzie informowany poprzez wskazaną w inicjalizacji funkcję obsługi zdarzenia *OnImageAvailableListener* [5].
C. Uzyskanie dostępu do kamery poprzez wywołanie *openCamera()* z określeniem numeru identyfikacyjnego urządzenia oraz obiektu wywołań zwrotnych związanych ze zmianą stanu pracy urządzenia.

W ramach klasy *RPiCamera* zdefiniujmy również obiekt wywołań zwrotnych, informujących nas o poprawnym uzyskaniu dostępu do modułu kamery (*onOpened*), jej odłączeniu (*onDisconnected*), wystąpieniu błędu (*onError*) oraz zamknięciu urządzenia (*onClosed*), co zaprezentowano na **listingu 3**.

Tym samym przygotowano szkielet klasy *RPiCamera*, który umożliwia nam poprawną inicjalizację modułu. Do wykorzystania funkcjonalności kamery, zatem brakuje metody umożliwiającej wyzwolenie przechwytywania obrazu z urządzenia. Funkcjonalność ta została zaimplementowana w metodzie *takePicture()* pokazanej na **listingu 4**.

Proces przechwytywania obrazu z modułu kamery rozpoczynamy od utworzenia sesji *CameraCaptureSession* [6], poprzez wywołanie metody *createCaptureSession()*. W wywołaniu tym określamy powierzchnię (*Surface*), do której będzie realizowany zapis (w omawianym przypadku jest to powierzchnia uprzednio skonfigurowanego obiektu *ImageReader*) oraz wskazujemy obiekt wywołań zwrotnych informujący nas

Czytelnicy którzy mają za sobą doświadczenia z oprogramowaniem obsługi kamery w systemie Android na urządzenia mobilne, mogą czuć się uprzywilejowani – dzięki pełnej integracji sterowników, obsługa dedykowanego modułu kamery Raspberry Pi w systemie Android Things jest analogiczna do obsługi kamer w urządzeniach mobilnych.

Listing 2. Inicjalizacja modułu kamery

```
private static final int IMAGE_WIDTH = 640;
private static final int IMAGE_HEIGHT = 480;

private CameraDevice mCameraDevice;
private CameraCaptureSession mCaptureSession;
private ImageReader mImageReader;

public void initializeCamera(Context context,
                             Handler backgroundHandler,
                             ImageReader.OnImageAvailableListener imageAvailableListener) {

    /* [A] Discover the camera instance */
    CameraManager manager = (CameraManager) context.getSystemService(CAMERA_SERVICE);
    String[] camIds = {};
    try {
        camIds = manager.getCameraIdList();
    } catch (CameraAccessException e) {
        Log.d(TAG, „Cam access exception getting IDs", e);
    }
    if (camIds.length < 1) {
        Log.d(TAG, „No cameras found");
        return;
    }
    String id = camIds[0];
    Log.d(TAG, „Using camera id „ + id);

    /* (B) Initialize the image processor */
    mImageReader = ImageReader.newInstance(IMAGE_WIDTH, IMAGE_HEIGHT, ImageFormat.JPEG, 1);
    mImageReader.setOnImageAvailableListener(imageAvailableListener, backgroundHandler);

    /* (C) Open the camera resource */
    try {
        manager.openCamera(id, mStateCallback, backgroundHandler);
    } catch (CameraAccessException cae) {
        Log.d(TAG, „Camera access exception", cae);
    }
}
```

Listing 3. Obiekt wywołań zwrotnych do monitorowania stanu modułu kamery

```
private final CameraDevice.StateCallback mStateCallback = new CameraDevice.StateCallback() {
    @Override
    public void onOpened(CameraDevice cameraDevice) {
        Log.d(TAG, „Opened camera.");
        mCameraDevice = cameraDevice;
    }

    @Override
    public void onDisconnected(CameraDevice cameraDevice) {
        Log.d(TAG, „Camera disconnected, closing.");
        cameraDevice.close();
    }

    @Override
    public void onError(CameraDevice cameraDevice, int i) {
        Log.d(TAG, „Camera device error, closing.");
        cameraDevice.close();
    }

    @Override
    public void onClosed(CameraDevice cameraDevice) {
        Log.d(TAG, „Closed camera, releasing");
        mCameraDevice = null;
    }
};
```

Listing 4. Obiekt wywołań zwrotnych do monitorowania stanu modułu kamery

```
public void takePicture() {
    if (mCameraDevice == null) {
        Log.w(TAG, „Cannot capture image. Camera not initialized.");
        return;
    }
    try {
        mCameraDevice.createCaptureSession(
                Collections.singletonList(mImageReader.getSurface()),
                mSessionCallback, null);
    } catch (CameraAccessException cae) {
        Log.d(TAG, „access exception while preparing pic", cae);
    }
}
```

Listing 5. Kod funkcji triggerImageCapture()

```
private void triggerImageCapture() {
    try {
        final CaptureRequest.Builder captureBuilder =
                mCameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_STILL_CAPTURE);
        captureBuilder.addTarget(mImageReader.getSurface());
        captureBuilder.set(CaptureRequest.CONTROL_AE_MODE, CaptureRequest.CONTROL_AE_MODE_ON);
        Log.d(TAG, „Session initialized.");
        mCaptureSession.capture(captureBuilder.build(), mCaptureCallback, null);
    } catch (CameraAccessException cae) {
        Log.d(TAG, „camera capture exception");
    }
}
```

o stanie utworzonej sesji. Poprawność utworzenia, aktywność i gotowość sesji do rozpoczęcia procedury przechwytywania obrazu, jest sygnalizowana poprzez wywołanie *onConfigured()*. Finalna postać funkcji konfigurującej i wyzwalającej przechwytywanie obrazu – *triggerImageCapture()* – została przedstawiona na **listingu 5**.

Posiadając kompletny kod klasy *RPiCamera*, możemy uzupełnić implementację głównej aktywności *MainActivity* o inicjalizację i realizację przechwytywania obrazu. Inicjalizacja kamery zostanie przeprowadzona w metodzie *onCreate()*, natomiast wywołanie metody *takePicture()*, nastąpi po przyciśnięciu dołączonego do wyprowadzenia *BCM23* przycisku. Korzystając z informacji przedstawionych w poprzednich częściach artykułu, dodajmy do klasy *MainActivity* obsługę przycisku i wyzwolenie przechwytywania obrazu, jak na **listingu 6**.

Wciśnięcie przycisku *BCM23* rozpoczyna sesję przechwytywania obrazu, podczas której dane z kamery przesyłane są do *ImageReader*'a. Zakończenie tego procesu oraz dostępność przechwyconych danych w postaci pliku JPEG jest sygnalizowana poprzez wywołanie metody *OnImageAvailableListener*. Przechwycony obraz jest przekazywany do funkcji *updateGUI()*, której zadaniem jest aktualizacja interfejsu użytkownika, według zleconych przez „klienta" założeń (funkcje związane z budową GUI nie będą omawiane w tym artykule – pełny kod źródłowy projektu jest dostępny do pobrania pod linkiem umieszczonym na końcu artykułu). Finalny efektu działania projektu został przedstawiony na filmie dostępnym pod adresem ***https://goo.gl/6ezmQ1***.

Personalizowana animacja startowa

Po wygenerowaniu pliku aplikacji *APK*, możemy przystąpić do personalizacji obrazu systemu *Android Things*. Jednym z elementów tej personalizacji, będzie zamiana animacji startowej – wyświetlającej domyślne logo systemu *Android Things* (**rysunek 2**), na statyczny obraz z pliku graficznego w formacie PNG.

Format animacji startowej został ściśle zdefiniowany przez dokumentację systemu operacyjnego *Android* [7]. Plik z animacją w postaci archiwum ZIP (*bootanimation.zip*), powinien zawierać plik *desc.txt* (z określonym formatem opisu animacji) oraz katalogi od *part0* do *partN*, zawierające kolejne „klatki" animacji zapisanych w postaci plików PNG.

Pierwsza linia opisu zawarta w pliku *desc.txt* definiuje następujące parametry animacji:
WIDTH HEIGHT FPS
gdzie:

- *WIDTH* – szerokość animacji wyrażona w pikselach,
- *HEIGHT* – wysokość animacji wyrażona w pikselach,
- *FPS* – liczba klatek na sekundę z jaką będzie wyświetlana animacja.

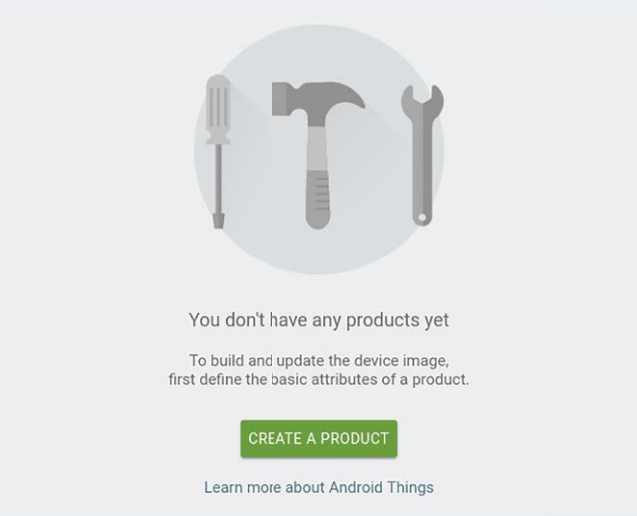
Rysunek 2. Domyślna animacja startowa systemu Android Things

Kolejne linie pliku *desc.txt* definiują poszczególne animacje, jakie będą wyświetlane na ekranie startowym (z uwzględnieniem kolejności zawartej w pliku). Pojedyncza linia opisuje przyjmuje następującą postać
TYPE COUNT PAUSE PATH [#RGBHEX]
gdzie:

- pole *TYPE* określa sposób wyświetlania danej animacji. Pole to może przyjmować jedną z dwóch wartości literowych:
  - *p* – wyświetlanie animacji może zostać przerwane w momencie gdy zakończony zostanie proces startu systemu,
  - *c* – animacja nie zostanie przerwana przez zakończony proces startu systemu,
- pole *COUNT* określa ile razy ma zostać odtworzona dana animacja – wartość 0 określa odtwarzanie animacji w pętli,
- *PAUSE* – pole określa liczbę klatek pauzy po zakończeniu danej animacji,
- *PATH* – pole wskazuje ścieżkę do katalogu zawierającego odpowiednio posortowane pliki PNG z animacją – np. *part0*, *part1*, itd.
- *RGBHEX* – opcjonalny parametr określający kolor tła (w formacie *#RRGGBB*).

Listing 6. Kod klasy MainActivity uzupełniony o obsługę modułu kamery

```
public class MainActivity extends Activity
{

    private Gpio button;
    private RPiCamera camera;
    private Handler cameraHandler;
    private HandlerThread cameraThread;

    /.../
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /.../

        /* configure button */
        PeripheralManagerService service = new PeripheralManagerService();
        try {
            button = service.openGpio("BCM23");
            button.setDirection(Gpio.DIRECTION_IN);
            button.setEdgeTriggerType(Gpio.EDGE_FALLING);
            button.registerGpioCallback(ButtonCallback);
        } catch (IOException e) {
            Log.e(TAG, "PeripheralIO API ERROR", e);
        }

        /.../

        /* create new handler and associated thread for camera */
        cameraThread = new HandlerThread("CameraBackground");
        cameraThread.start();
        cameraHandler = new Handler(cameraThread.getLooper());

        /* initialize camera */
        camera = RPiCamera.getInstance();
        camera.initializeCamera(this, cameraHandler, mOnImageAvailableListener);
    }
    /*
     * Button Callback
     */
    private GpioCallback ButtonCallback = new GpioCallback() {
        @Override
        public boolean onGpioEdge(Gpio gpio) {

            camera.takePicture();
            return true;
        }
    };

    /*
     * Listener for new camera images
     */
    private ImageReader.OnImageAvailableListener mOnImageAvailableListener =
            new ImageReader.OnImageAvailableListener() {
                @Override
                public void onImageAvailable(ImageReader reader) {

                    Image image = reader.acquireLatestImage();

                    ByteBuffer imageBuf = image.getPlanes()[0].getBuffer();
                    final byte[] imageBytes = new byte[imageBuf.remaining()];
                    imageBuf.get(imageBytes);
                    image.close();

                    updateGUI(imageBytes);
                }
            };
  /.../
}
```



Rysunek 3. Utworzenie nowego produktu w ramach Android Things Console

Poprzez odpowiednią konfigurację pliku *desc.txt*, użytkownik mam możliwość tworzenia bardziej rozbudowanych, kilku etapowych animacji, złożonych z animacji startowej (wyświetlanej raz) oraz animacji odtwarzanej w pętli, aż do zakończenia procesu uruchamiana systemu. Przykład takiej konfiguracji pliku *desc.txt*, został przedstawiony poniżej:

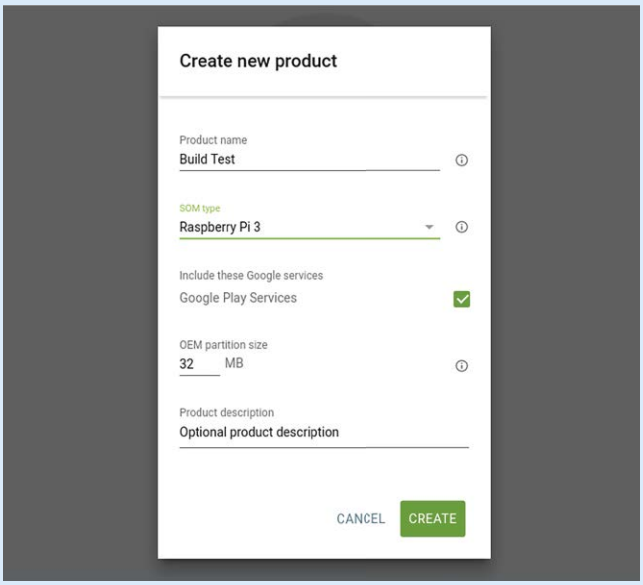
800 480 30
p 1 0 part0
p 0 0 part1

(animacja o parametrach *800×480@30* jest dwuetapowa – pierwszy etap, na który składają się pliki PNG z katalogu *part0*, jest wyświetlany raz, następnie w nieskończonej pętli – aż do momentu zakończenia procesu startu systemu – wyświetlana jest animacja z katalogu *part1*).

W realizowanym projekcie wykorzystany zostanie statyczny obraz z pliku PNG o rozdzielczości 1200×500 pikseli. Konfiguracja z pliku *desc.txt*, będzie wówczas prezentowała się następująco:

1200 500 1
p 0 0 part0

Ważnym zagadnieniem przy tworzeniu własnej animizacji startowej jest również sposób utworzenia archiwum *bootanimation.zip*. Pliki umieszczone w ramach pliku ZIP nie mogę być skompresowane, lecz powinny być umieszczone w archiwum w tzw. trybie *„store only”*. Utworzenie takiego archiwum z linii poleceń systemu *Linux*, może zostać zrealizowane następująco (gdzie flaga -0 wymusza tryb *„store only”*) – *zip -0qry -i *.txt *.png *.wav @ ../bootanimation.zip *.txt part**.



Rysunek 4. Konfiguracja nowego produktu w Android Things Console

Budowa obrazu systemu z Android Things Console

Mając przygotowany plik aplikacji użytkownika *APK* oraz animację startową w postaci pliku *bootanimation.zip*, czas przystąpić do budowy obrazu. Do tego celu firma Google przygotowała narzędzie *Android Things Console*, dostępne pod adresem ***https://goo.gl/dhvAhG***. Narzędzie *Android Things Console* umożliwia:

- budowę personalizowanych obrazów systemu *Android Things*, zawierających zintegrowaną aplikację użytkownika,
- zdalną aktualizację oprogramowania – aplikacji użytkownika i wersji systemu operacyjnego.

Po zalogowaniu się do aplikacji *Android Things Console* (z wykorzystaniem konta Google) i zaakceptowania warunków świadczenia usługi, użytkownik uzyska możliwość utworzenia nowego produktu, jak przedstawiono to na **rysunku 3**. W oknie konfiguracji nowego produktu (**rysunek 4**) użytkownik zostanie poproszony o:

- określenie nazwy produktu (jest to nazwa wewnętrzna projektu, która nie będzie widoczna dla użytkownika końcowego),
- wybranie platformy sprzętowej (z listy aktualnie wspieranych zestawów deweloperskich),
- włączenia/wyłącznie wsparcia dla usług *Google Play*,
- określenia wielkości partycji OEM na której będzie instalowana aplikacja użytkownika,
- opcjonalne podanie opisu nowo utworzonego produktu.

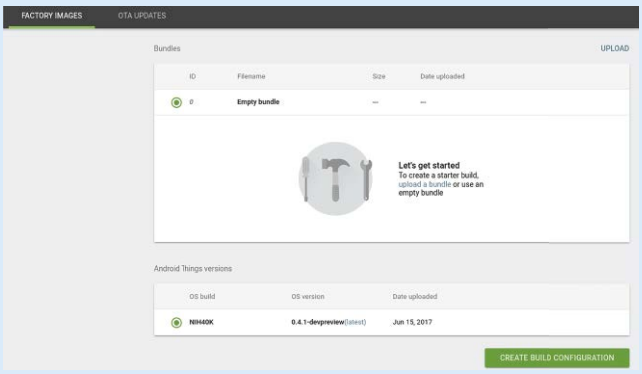
Po wybraniu przycisku *CREATE*, aplikacja przeniesie nas do zakładki *PRODUCT SETTINGS*, zawierającej podsumowanie konfiguracji utworzonego produktu.

Zakładka *FACTORY IMAGES* (**rysunek 5**) domyślnie podzielona została na dwie części:

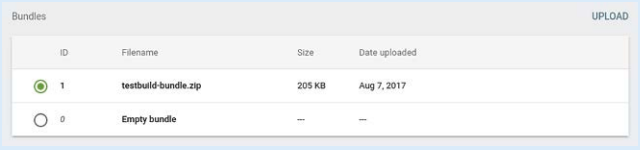
- *Bundles* – zawierającą informacje o utworzonych pakietach użytkownika, przeznaczonych do personalizacji systemu,
- *Android Things Versions* – umożliwiająca użytkownikowi wybranie wersji systemu operacyjnego *Android Things*.

W domyślnej konfiguracji (bez utworzonego pakietu personalizującego system), po wybraniu przycisku *CREATE BUILD CONFIGURATION*, użytkownik ma możliwość zbudowania podstawowej wersji obrazu systemu – w postaci, którą można również pobrać pod adresem ***https://goo.gl/8nQ5gp***.

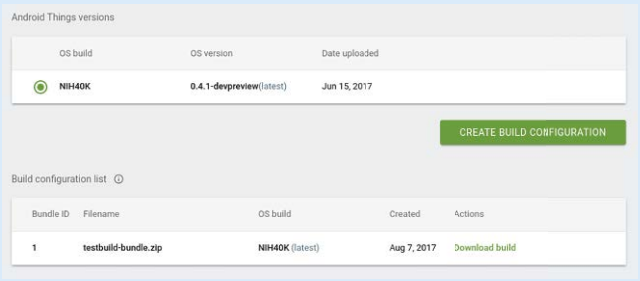
Naszym zadaniem jest przygotowanie wersji systemu, która będzie domyślnie zawierała przygotowaną aplikację użytkownika



Rysunek 5. Zakładka Factory Images



Rysunek 6. Utworzenie nowego pakietu w sekcji „Bundles”



Rysunek 7. Lista obrazów utworzonych za pomocą narzędzia Android Things Console

oraz personalizowaną animację startową. Po wybraniu przycisku *UPLOAD* w sekcji *Bundles*, użytkownik zostanie poproszony o wskazanie archiwum ZIP zawierającego dane dla partycji OEM. Archiwum ZIP służące do personalizacji systemu powinno zawierać następujące pliki:

- *bootanimation.zip* – [plik nieobowiązkowy] animacja użytkownika przygotowana zgodnie z wytycznymi z podrozdziału „*Personalizowana animacja startowa*”,
- *<user-space driver.apk>* – [plik nieobowiązkowy] sterowniki działające w przestrzeni użytkownika jako serwis systemowy,
- *<main.apk>* – [plik obowiązkowy] aplikacja użytkownika definiująca główny punkt wejścia (*action=MAIN, category=IOT_LAUNCHER*),
- *<sub.apk>* – [plik nieobowiązkowy] dowolne inne aplikacje użytkownika, które mogą zostać uruchomione przez aplikację główną.

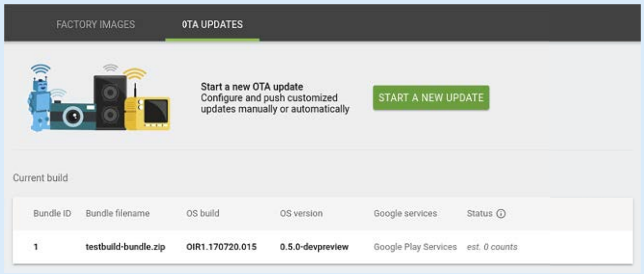
Utwórzmy zatem nowe archiwum ZIP (tym razem jest to standardowy plik ZIP w odróżnieniu od trybu „*store only*” dla animacji startowej) zawierające plik *APK* oraz *bootanimation.zip*. Po załadowaniu nowego pakietu, sekcja „*Bundles*” zostanie zaktualizowana, jak przedstawiono to na **rysunku 6**.

W ostatnim kroku – wybierając z sekcji „*Bundles*” nasz nowo utworzony pakiet, a następnie z sekcji *Android Things Versions* ostatnią dostępną wersje systemu – utwórzmy spersonalizowany obraz dla naszego urządzenia. Lista wszystkich zbudowanych obrazów oraz odnośników do ich pobrania, zostanie wyświetlona w sekcji „*Build configuration list*” – **rysunek 7**.

Aktualizacje OTA

W naszym hipotetycznie realizowanym scenariuszu, udało nam się dostarczyć 100 urządzeń „inteligentnych dzwonków” dla naszego klienta. Nie trudno wyobrazić sobie jednak sytuację, gdzie przygotowane oprogramowania posiada błąd, który udało się odtworzyć dopiero po kilku miesiącach ciągłego działania urządzenia. W takich przypadkach, bardzo często koszty związane z dostarczeniem urządzeń do producenta i wgraniem nowego oprogramowania mogą okazać się bardzo wysokie. Z pomocą mogą przyjść aktualizacje *OTA* (*Over The Air*), czyli mechanizm aktualizacji zdalnych. Dla obrazów zbudowanych z wykorzystaniem *Android Things Console*, firma Google udostępnia infrastrukturę umożliwiającą przeprowadzenie zdalnej aktualizacji urządzeń (pakietu użytkownika i wersji systemu *Android Things*).

Aby przeprowadzić zdalną aktualizację, w systemie *Android Things Console*, przechodzimy do zakładki *OTA UPDATES*.



Rysunek 8. Aktualizacja systemu poprzez Android Things Console i mechanizm OTA

Klikając przycisk *START A NEW UPDATE*, zostaniemy poproszeni o konfigurację nowo przygotowywanej aktualizacji. Panel konfiguracji jest tożsamy z zakładką *FACTORY IMAGES*. Użytkownik ma możliwość utworzenia nowego pakietu ze zaktualizowaną wersją aplikacji lub wgrania najnowszej wersji systemu *Android Things* (obie operacje mogą zostać przeprowadzone w ramach jednej aktualizacji). Proces aktualizacji rozpoczyna wybranie przycisku *PUSH UPDATE*. Od tego momentu, nowa aktualizacja jest dostępna do pobrania dla wszystkich urządzeń końcowych. Serwis *update_engine* uruchomiony w systemie *Android Things* sprawdza dostępność zdalnych aktualizacji co 300 minut, tak więc proces aktualizacji nie odbędzie się na wszystkich urządzeniach jednocześnie i może potrwać kilka godzin. *Android Things Console* udostępnia informacje o aktualnym postępie aktualizacji wszystkich urządzeń, jak pokazano na **rysunku 8**.

W czasie powstawania ostatniego odcinka z serii poświęconej systemowi Android Things, zgodnie z przyjętą polityką (regularne aktualizacje systemu w odstępach 6-8 tygodni), firma Google udostępniła najnowszą aktualizację systemu o numerze 5 (oznaczenie kodowe: OIR1.170720.015). Do najważniejszych zmian należą:

- zmiana wersji bazowej systemu do Android O (API 26),
- wsparcie dla nowej platformy deweloperskiej – NXP SprIoT i.MX6UL,
- włączenie wsparcia dla OpenGL ES 2.0,
- sprzętowe wsparcie dla platform wyposażonych w GPU (w tym Raspberry Pi 3),
- dodanie nowego API – DeviceManager,
- możliwość dynamicznej konfiguracji zadań pełnionych przez wyprowadzenia GPIO (Dynamic Pin Muxing).

Wraz z najnowszą aktualizacją, obrazy systemu dostępne są do pobrania wyłączenie poprzez narzędzie Android Things Console. Opis pełnej listy zmian dostępny jest pod adresem **https://goo.gl/6aW2jV**.

Łukasz Skalski

Linki zewnętrzne:
https://goo.gl/QN2yRV
https://goo.gl/e5Gp6s
https://goo.gl/Y7DobF
https://goo.gl/63bF3L
https://goo.gl/pZ3zU7
https://goo.gl/wgNWeo
https://goo.gl/fnUqMv