Autori: Marco Alessio, Pietro Prandini e Riccardo Ertolupi

android things

Programmazione dei Sistemi Embedded

Università Degli Studi Di Padova - a.a. 2017-2018

Indice

Capitolo 1:
Introduzione ad Android Things

1.1 Cos'è Android Things?

Android Things è una piattaforma software specifica per sistemi embedded sviluppata da Google. Essa, infatti, nasce con lo specifico scopo di permettere la produzione di prodotti professionali sviluppati su una solida piattaforma software senza la necessità di aver acquisito particolari conoscenze riguardo la progettazione di sistemi embedded.

Tale piattaforma è in continuo sviluppo e il suo ecosistema è in fase di espansione grazie al coinvolgimento degli sviluppatori tramite guide, repository git, blog ed esempi di codice. Risulta importante anche il supporto di hardware sia di sviluppo che di produzione e la Console di Android Things che permette di costruire e aggiornare immagini del sistema specifiche per i propri prodotti.

Al momento della stesura di questo report, la versione di Android Things è la 1.0 e come piattaforme hardware supportate presenta alcuni System-On-Modules come NXP i.MX8M, Qualcomm SDA212, Qualcomm SDA624 e MediaTek MT8516 per la produzione, mentre altre schede per lo sviluppo come NXP Pico i.MX7D e Raspberry Pi 3 Model B per le quali sono disponibili anche degli starter-kit ufficiali.

Nel proseguo del report si analizzeranno nello specifico gli aspetti di Android Things 1.0 e tale piattaforma verrà utilizzata per lo sviluppo di un'applicazione esemplificativa.

1.2 Differenze da Android

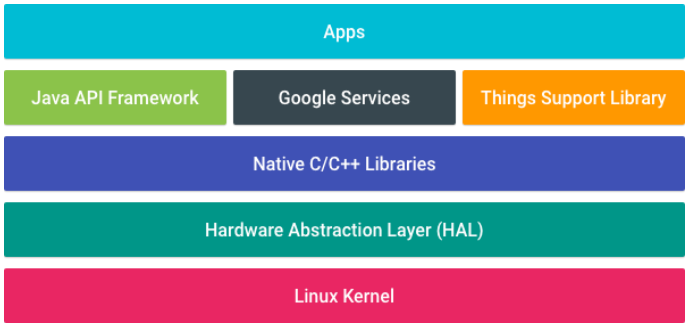
Android Things si differenzia dalla piattaforma Android specifica per smartphones e tablets per alcuni aspetti: innanzitutto perché è ottimizzata per dispositivi embedded i quali, non essendo smartphones o tablets, non hanno necessariamente alcuni componenti come, per esempio, lo schermo e il metodo di input touch che, invece, sono richiesti per la versione mobile di Android. Per questo motivo non sono supportate alcune API relative all'interfaccia grafica (NotificationManager, KeyguardManager, Wallpaper Manager) e altre specifiche del mondo Android mobile (SpeechRecognizer, FingerprintManager, NfcManager, SmsManager,

TelephonyManager, UsbAccessory, WifiAwareManager, AppWidgetManager, AutofillManager, BackupManager, CompanionDeviceManager, Activity Picture-in-picture, PrintManager, SipManager).

Oltre alle API già citate, non sono inclusi la suite standard di app di sistema e content provider. In particolare, si devono evitare le API dei seguenti content providers: CalendarContract, ContactsContract, DownloadManager, MediaStore, Settings, Telephony, UserDictionary, VoicemailContract.

Per verificare se qualche API è disponibile si può utilizzare il metodo hasSystemFeature() della classe PackageManager.

Inoltre, Android Things supporta solo un sottoinsieme delle API Google per Android.



Android Things è compatibile con Android NDK, il toolset per implementare parti di app in codice nativo tramite linguaggio C/C++, ma, a differenza di Android mobile, le librerie native devono essere mantenute all'interno dell'APK a runtime. La motivazione di questa scelta sta nel fatto che solitamente l'hardware per cui è pensato ha una memoria limitata.

Un'altra importante caratteristica riguarda la metodologia di aggiornamento del sistema definita over-the-air (OTA): viene effettuata tramite la Console ufficiale di Android Things la quale verrà analizzata nello specifico più avanti nel report.

Vi sono API specifiche per la connettività. Android Things infatti aggiunge particolari funzionalità alle API riguardanti il Bluetooth ed è prevista una connessione LoWPAN (Low Power Wireless Personal Area Networks).

Infine, sono previste API specifiche per l'interazione con le periferiche I/O, che permettono la comunicazione con sensori e periferiche utilizzanti protocolli e interfacce standard. È possibile anche estendere il framework dei servizi con driver personalizzati.
Più precisamente, sono disponibili API per la gestione di GPIO, PWM, I2C, SPI e UART.

1.3 Avvio automatico dell'app

In sostanza un dispositivo Android Things è pensato per svolgere i compiti di una sola applicazione e tale applicazione deve essere disponibile all'utente sempre e già dal momento successivo all'accensione del dispositivo. In altre parole, il sistema si aspetta che vi sia un'applicazione che presenta una activity definita di home da lanciare successivamente all'avvio del sistema. Inoltre, risulta utile che tale applicazione si avvii nuovamente in caso di crash. Queste proprietà vengono garantite tramite intent nel Manifest dichiarando **`android.intent.category.HOME`** e **`android.intent.category.DEFAULT`** nella sezione di intent-filter dell'activity.

In caso contrario, viene avviata un'applicazione di default che mostra alcune informazioni sul sistema e permette di effettuare alcune impostazioni, come connettersi alla rete.

Di seguito si può vedere un esempio del file Manifest che implementa la funzionalità di avvio automatico dopo il boot:

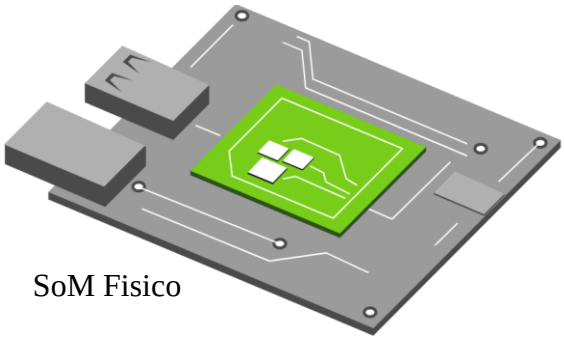
```
<application>
    <uses-library android:name="com.google.android.things"/>
    <activity android:name=".HomeActivity">

        <!-- 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
            and re-launch if the app terminates.
        -->
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name=
                "android.intent.category.HOME"/>
            <category android:name=
                "android.intent.category.DEFAULT"/>
        </intent-filter>

    </activity>
</application>
```

1.4 Hardware supportato

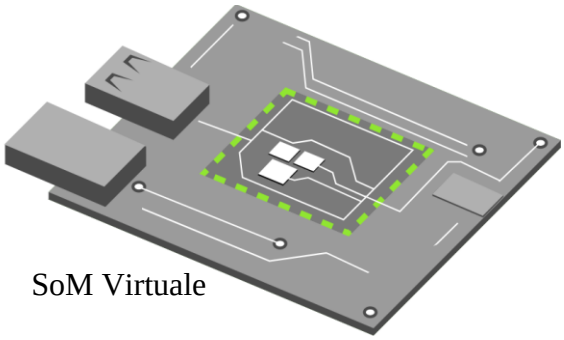
Come introdotto in precedenza sono supportate varie piattaforme hardware e sono suddivise in quelle adibite per la produzione e quelle per lo sviluppo.

Per quanto riguarda i System-on-Modules destinati alla produzione, Google assicura aggiornamenti di sicurezza per un minimo di tre anni e, inoltre, vi certifica requisiti di sicurezza quali Key and ID Attestation per la creazione, la memorizzazione e l'uso di chiavi crittografate e Verified Boot per il controllo dell'integrità del sistema.

Tra i SoM supportati v'è una differenza a cui prestare attenzione: ovvero possono essere fisici o virtuali. Un SoM fisico è un modulo concreto integrabile in un prodotto finale come un singolo componente, mentre un SoM virtuale necessita di essere progettato e realizzato secondo le specifiche previste e può essere inserito all'interno della board del prodotto. Generalmente i SoM virtuali vengono usati per dispositivi prodotti in grandi dimensioni, dove il costo di tale realizzazione viene compensato da una maggiore flessibilità di progettazione.



SoM Fisico



SoM Virtuale

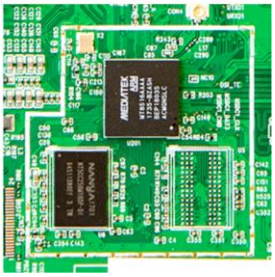
I SoM per la produzione supportati al momento sono NXP i.MX8M (fisico), Qualcomm SDA212 (fisico), Qualcomm SDA624 (fisico) e MediaTek MT8516 (virtuale).

NXP i.MX8M

Qualcomm SDA212

Qualcomm SDA624

MediaTek MT8516



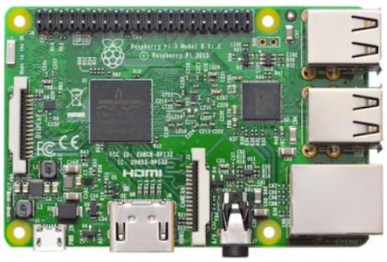
In generale questi dispositivi hanno caratteristiche simili:

- Hanno tutti un processore ARM che va dal quadcore a 1.267 Ghz del Qualcomm SDA212 al octacore a 1.8 Ghz del Qualcomm SDA624.
- La memoria RAM va dai 512 MB del MediaTek MT8516 ai 2GB del Qualcomm SDA624.

- Sono tutti muniti di periferiche UART, I2C, SPI, PWM e GPIO.
- Hanno tutti sia il Wi-Fi 802.11ac che il Bluetooth 4.2. Inoltre, nel NXP i.MX8M e nel MediaTek MT8516 è presente una connessione 10/100/1000 Ethernet.
- Le dimensioni di questi dispositivi sono all'incirca di 25cm^2.

NXP Pico i.MX7D

Raspberry Pi 3 Model B



Sono state pensate anche due piattaforme per lo sviluppo: NXP Pico i.MX7D e Raspberry Pi 3 Model B. Tali piattaforme sono utili per realizzare prototipi e per eseguire test e non sono certificate con i requisiti di sicurezza quali quelli dei SoM per la produzione e potrebbero non ricevere aggiornamenti di sicurezza e stabilità. Nonostante questo, Google promette gli aggiornamenti necessari quanto meno per assicurare la possibilità di sviluppare su tali piattaforme e possibilmente portare il codice sulle piattaforme di produzione.

Queste piattaforme per lo sviluppo presentano le seguenti caratteristiche:

- Entrambe montano un processore ARM:
  - NPX: 1GHz dual-core ARM Cortex A7.
  - RPi: 1.2GHz quad-core ARM Cortex A53.
- RAM:
  - NXP: 512MB.
  - RPi: 1GB.
- Wi-Fi:
  - NPX: 802.11ac.
  - RPi: 802.11n.
- Bluetooth 4.1.
- Ethernet:
  - NPX: 10/100/1000.
  - RPi: 10/100.
- Hanno entrambi almeno una porta USB.
- Hanno la possibilità di connessione ad un display esterno, in particolare la RPi presenta una porta HDMI standard.

Inoltre si interfacciano con periferiche esterne con i metodi di comunicazione utilizzati anche dai SoM di produzione: UART, I2C, SPI, PWM e GPIO.
La NPX ha una memoria eMMC da 4GB, mentre la RPi ha uno slot per MicroSD card.

1.5 Sitografia del capitolo 1

1. Android Things - Releases - Android Things
   https://developer.android.com/things/
2. Android Things - Overview
   https://developer.android.com/things/get-started/
3. Android Things - Features and API
   https://developer.android.com/things/versions/things-1.0
4. Esempi Android Things
   https://developer.android.com/samples/?technology=iot
5. Repository git ufficiale di Android Things
   https://github.com/androidthings
6. Blog di Android Things
   https://developers.googleblog.com/search/label/Android%20Things
7. Console di Android Things
   https://partner.android.com/things/console/
8. Hardware supportato Android Things
   https://developer.android.com/things/hardware/

Capitolo 2:
Strumenti di sviluppo specifici

2.1 Connessione all'ambiente di sviluppo

La comunicazione tra la piattaforma hardware ed Android Studio avviene mediante il tool ADB (Android Debug Bridge), rilasciato assieme all'IDE. Di default, ADB viene installato alla seguente locazione:

- Windows: **`%USERPROFILE%\AppData\Local\Android\Sdk\platform-tools`**
- GNU/Linux: **`~/Android/Sdk/platform-tools/`**

Per connettere la propria scheda con ADB, aprire il terminale, posizionarsi alla corretta locazione, e digitare: **`adb connect <ip-address>`**

È necessario quindi stabilire un indirizzo IP valido per la propria scheda. La connessione fisica della propria scheda può essere fatta all'Access-Point della propria rete, oppure direttamente al pc di sviluppo. Sono possibili quattro modalità differenti con cui connettere la scheda alla propria rete:

- **Via cavo Ethernet all'Access-Point**: è il metodo suggerito da Google, oltre ad essere quello più semplice, tuttavia richiede l'accesso fisico all'Access-Point della rete, opzione che potrebbe non essere disponibile.
- **Via cavo seriale al pc di sviluppo**: consiste nel connettersi tramite UART alla scheda di sviluppo con un adattatore USB collegato al computer. È necessario utilizzare i seguenti parametri per la connessione UART:
  - Baud Rate: 115200
  - Data Bits: 8
  - Parity: no
  - Stop bits: 1

  Risulta essere utile solamente se non è possibile accedere ad ADB in altri modi o se non è disponibile nessuna connessione di rete.
- **Via cavo Ethernet al PC di sviluppo**: modalità utile se non si possono utilizzare altri metodi di connessione oppure, semplicemente, se risulta essere una soluzione comoda. È necessario che il PC assegni un indirizzo IP al dispositivo connesso, abilitando la funzionalità DHCP alla porta Ethernet del proprio PC.

- **Via hotspot su Windows 10**: nelle ultime versioni del sistema operativo di Microsoft è stata introdotta la funzionalità “Hostspot mobile”, che consente di condividere la propria connessione Internet con altri dispositivi, fino ad un numero massimo di 8. La sua particolarità consiste nel fatto che i dispositivi connessi vengono visti come appartenenti alla stessa rete del pc, e ricevono automaticamente un indirizzo IP.
  È il metodo migliore nei casi in cui ci si connette ad una rete Wi-Fi che abbia modalità di autenticazione non supportate, come la WPA2/Enterprise utilizzata da Eduroam. Risulta essere l’opzione più comoda su pc Windows ed è anche molto stabile.

2.1.1 La nostra esperienza usando la Raspberry Pi 3 B

Nel caso del nostro gruppo, abbiamo testato tutti i metodi descritti in precedenza, eccetto la comunicazione via porta seriale, poiché la UART è impegnata per comunicare con una periferica.

La connessione della Raspberry Pi via cavo all’Access-Point ha fornito risultati alterni: di norma ha funzionato correttamente, tuttavia in alcuni casi la connessione ha smesso di funzionare e ci ha costretto a riconnettere manualmente la scheda o anche a riavviarla.

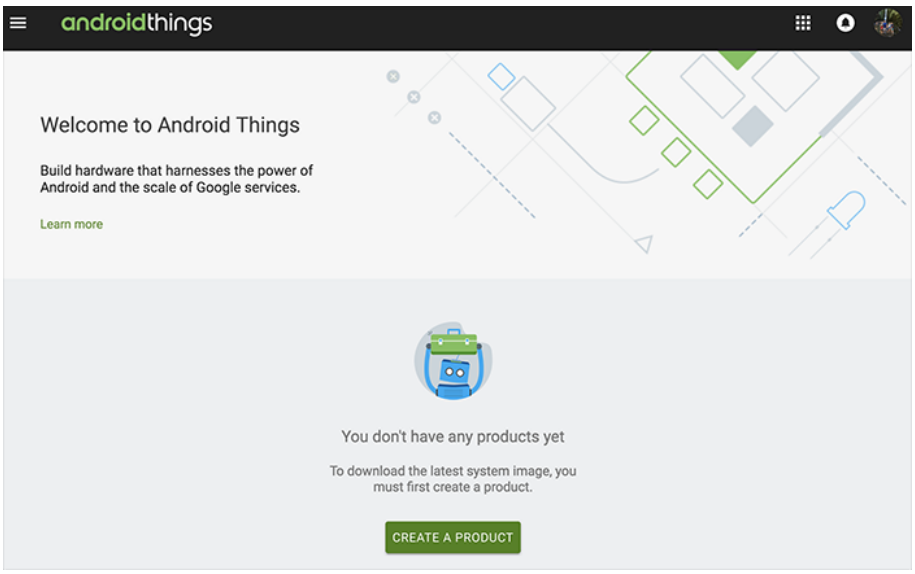
La connessione via cavo al pc di sviluppo è stato utilizzato su un sistema “Debian 9 Stretch” (al momento: “stable”), tramite l’installazione e la configurazione del pacchetto “isc-dhcp-server”, mantenuto dalla comunità Debian. Abbiamo appurato che questo sistema di connessione risulta essere sempre stabile ed affidabile, la connessione persiste fino ad esplicita chiusura da parte dello sviluppatore e ha sempre risposto positivamente ai comandi impartiti tramite ADB.

Il metodo via hotspot su Windows 10 è stato usato su più computer e si è rivelato essere il metodo migliore durante lo sviluppo in questi sistemi. È risultato particolarmente utile vista l’impossibilità di usare una connessione via cavo per la mancanza di una porta ethernet sui computer portatili più recenti. Al momento l’unico aspetto negativo di questo metodo è l’impossibilità di assegnare comodamente un indirizzo IP statico alla scheda.

2.2 Console

La console è lo strumento con cui creare e gestire i propri prodotti basati su Android Things. Tra le funzionalità offerte vi sono la possibilità di creare più modelli dello stesso prodotto, gestire gli aggiornamenti OTA (Over The Air), e monitorare l'andamento dei propri prodotti attraverso Analytics. Per poter usufruire della console è necessario possedere un account Google.

Si può accedere alla console dal link seguente: https://partner.android.com/things/console/



2.2.1 Creazione prodotto e modelli

Il primo passo è quello di definire un nuovo prodotto, definendone un nome, la piattaforma hardware di riferimento ed una eventuale descrizione. È possibile dopo la creazione associare altri sviluppatori al proprio prodotto, associando il loro account Google al progetto. Una volta creato il prodotto è possibile definire uno o più modelli dello stesso, varianti del medesimo prodotto basati sullo stesso hardware.

2.2.2 Creazione build

A un modello si possono associare diverse build, le quali permettono di personalizzare la versione di Android Things installata, gli applicativi e le risorse preinstallati. In particolare, oltre alle applicazioni create dallo sviluppatore, è possibile integrare le "Google Apps", applicazioni sviluppate e mantenute da Google, come ad esempio Google Play Services.

Nelle nuove applicazioni da inserire nella build è necessario garantire esplicitamente i permessi potenzialmente pericolosi richiesti dall'applicazione sul file **`Manifest.xml`** e, quando si inserisce l'app nella build, verrà data la possibilità allo sviluppatore di verificare e rimuovere alcuni permessi. Al contrario di Android mobile, all'utente finale il meccanismo dei permessi è completamente trasparente e non possono essere modificati.

Le risorse possono essere qualunque file che sia necessario al funzionamento del prodotto. Al momento, le risorse caricate dall'utente supportate sono estremamente limitate: solo il file **`bootanimation.zip`**, che definisce l'animazione da mostrare sul display, e il file audio di sottofondo, eseguito durante l'avvio del prodotto. Eventuali altri file necessari dovranno necessariamente essere integrati nell'APK dell'applicazione.

Penultimo step è quello della configurazione delle periferiche hardware. Dalla versione 1.0 di Android Things, l'uso delle periferiche GPIO, I2C, UART, SPI, PWM è consentito solo previa indicazione al momento della creazione della build.
Nella configurazione delle periferiche GPIO, è necessario impostare il pin utilizzato e se si vuole usare delle resistenze integrate nel pin. È possibile scegliere tra i seguenti tipi di resistenza:

- **Nessuna**: nessuna resistenza viene abilitata.
- **Pull-up**: abilita una resistenza collegata tra il GPIO e il livello di tensione alto.
- **Pull-down**: abilita una resistenza collegata tra il GPIO e GND.

Generalmente, si usa nessuna resistenza per i GPIO usati come output, mentre si usano le resistenze di pull-up o pull-down per i GPIO usati come input così, nel caso non fosse collegato nulla, il pin non si trova in uno stato indeterminato.
Per le altre periferiche, è necessario indicare solamente il bus usato, con la sola eccezione di I2C in cui è necessario indicare anche la frequenza di trasmissione dati.

Infine, è possibile personalizzare il partizionamento dello storage dedicato del prodotto. Lo spazio di archiviazione, limitato a massimo 4 GB, viene suddiviso in 3 partizioni: la prima è dedicata al sistema operativo Android Things, di circa 1.3 GB. La seconda è dedicata al prodotto, con uno spazio a disposizione massimo di 1122 MB, mentre l'ultima è dedicata ai dati utente, ed occupa la parte restante dello spazio a disposizione.

Alla fine della creazione di una build è possibile associarci dei tag testuali e impostare il flag "Factory Image", il quale marca la build come base affinché le build successive siano delle immagini incrementali rispetto ad essa (ossia vengono memorizzate solo le differenze tra la build e la build base).

Finito il processo di creazione della build, essa è disponibile al download nelle due versioni Production o Development. La prima è destinata al rilascio al pubblico, mentre la seconda è destinata a soli fini di sviluppo. Le due versioni differiscono per l'accesso agli strumenti di debug, come ad esempio ADB, disponibili solamente nella versione Development.

2.2.3 Gestione aggiornamenti

Dopo aver selezionato un modello, nella sezione "Release" è possibile rilasciare aggiornamenti OTA (Over The Air) per il modello prescelto. Esistono quattro canali di aggiornamento creati di default (Beta, Canary, Development, Stable), tuttavia è possibile definirne di propri. Ogni dispositivo è sempre associato ad un solo canale, di default canale stable.

Per rilasciare un nuovo aggiornamento, nella Console selezionare un canale di aggiornamento e successivamente "Start a new update". Quindi è necessario scegliere una build e la percentuale di dispositivi che riceveranno immediatamente l'update. Se la percentuale è inferiore al 100%, alcuni dispositivi sono scelti a caso per installare l'aggiornamento, mentre il resto dei dispositivi lo ignorerà e il roll-out degli aggiornamenti precedenti continua. È possibile visionare la percentuale e il numero assoluto di dispositivi a cui è stato installato correttamente l'aggiornamento, così come bloccare il roll-out dello stesso nel caso sia necessario.

Esiste una funzione di Auto-Update di un canale: la console automaticamente pubblica, per tutti i dispositivi iscritti a quel canale, gli aggiornamenti di sicurezza e gli update "minor version" di Android Things. I dispositivi non verranno aggiornati automaticamente alle successive "major version", pertanto sarà necessario intervenire tramite la pubblicazione di un nuovo update OTA tramite la console. Tutte le build create dalla funzionalità di Auto-Update sono visibili nella lista delle build e presentano il tag "auto-update". Il processo di auto-aggiornamento avviene così: dopo 7 giorni inizia il roll-out al 10% dei dispositivi e dopo 14 giorni viene reso disponibile a tutti. Requisito necessario affinché la funzionalità sia attiva è che sia presente un aggiornamento OTA per quel specifico canale.

2.2.4 Analytics

Infine, la funzionalità di Analytics mostra delle statistiche inerenti ai propri prodotti basati su Android Things. Al momento è possibile analizzare le seguenti variabili:

- Controllo, download e installazione degli update e successivo reboot.

- Errori nel processo di aggiornamento.
- Accensioni dei dispositivi.
- Dispositivi in uso.

Le statistiche possono essere filtrate per periodo, prodotto e modelli, versione del sistema operativo, canale di aggiornamento e nazione.

2.3 Sitografia del capitolo 2

1. Android Things - Hardware - Raspberry Pi
   https://developer.android.com/things/hardware/raspberrypi
2. Debian - Configurazione DHCP server
   https://wiki.debian.org/DHCP_Server

3. Android Things - Console - Overview
   https://developer.android.com/things/console/
4. Android Things - Console - Create an Android Things product
   https://developer.android.com/things/console/create
5. Android Things - Console - Configure an Android Things product
   https://developer.android.com/things/console/configure
6. Android Things - Console - Create a build for an Android Things product
   https://developer.android.com/things/console/build
7. Android Things - Console - Manage Apps
   https://developer.android.com/things/console/manage-apps
8. Android Things - Console - Push an update for an Android Things product
   https://developer.android.com/things/console/update
9. Android Things - Console - Monitor analytics
   https://developer.android.com/things/console/analytics

Capitolo 3:
Aggiornamenti Over-The-Air

3.1 Device Update API

Questa API permette di gestire la ricerca e l'installazione di nuovi aggiornamenti OTA in una applicazione Android Things. Il processo di aggiornamento di un dispositivo avviene in tre passi:

- Controllo della disponibilità di nuovi aggiornamenti software.
- Download e installazione dell'aggiornamento.
- Reboot del dispositivo con la nuova versione del software.

A ogni dispositivo è associata una policy, che gestisce le modalità e i tempi di installazione degli aggiornamenti. Il servizio di gestione degli aggiornamenti adeguerà il proprio operato in base alla policy impostata.

Per il suo uso è necessario indicare i seguenti permessi nel manifest dell'applicazione:

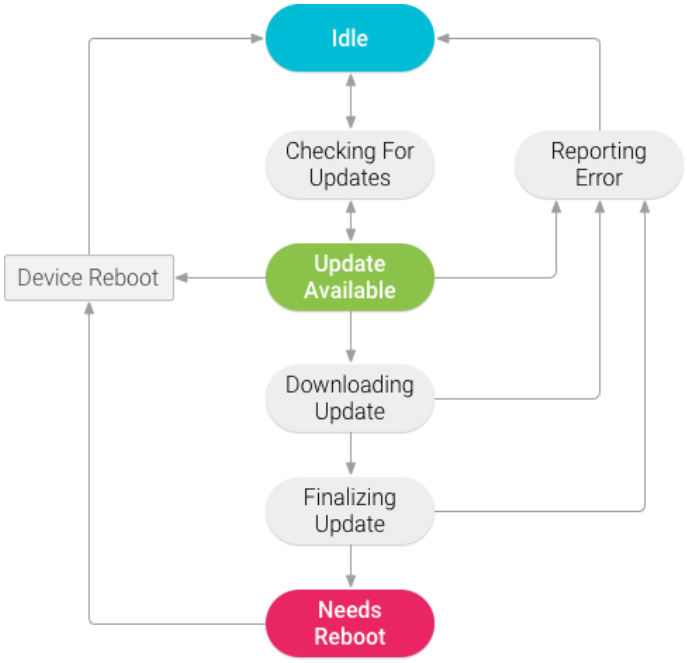
```
<uses-permission android:name =
    "com.google.android.things.permission.MANAGE_UPDATE_POLICY" />

<uses-permission android:name =
    "com.google.android.things.permission.PERFORM_UPDATE_NOW" />
```

3.2 Stato degli aggiornamenti

Un dispositivo si può trovare in uno dei seguenti stadi:

- **Idle**: lo stato di default di un dispositivo. Il servizio di gestione aggiornamenti controllerà periodicamente se un nuovo aggiornamento è stato rilasciato attraverso la Console. possibile forzare il controllo via codice, ad esempio all'avvio del dispositivo.
- **Update Available**: un aggiornamento è disponibile, ma non è ancora partito il download dello stesso. Il comportamento del servizio di aggiornamento dipenderà dall'update policy impostato per il dispositivo.
- **Downloading Update**: è in corso il download dell'aggiornamento.
- **Finalizing Update**: è finito il download ed è in corso l'installazione dell'aggiornamento.

- **Updated Needs Reboot**: l'aggiornamento è stato installato, tuttavia è necessario il riavvio del dispositivo per finire la procedura di aggiornamento.
- **Reporting Error**: un errore è avvenuto durante il processo di aggiornamento.



La presenza nella policy di una deadline garantisce che alla sua scadenza, qualora la procedura di aggiornamento non sia stata completata, venga ultimata automaticamente.

3.3 La classe UpdateManager

La classe **`UpdateManager`** è il punto focale della API per la gestione degli aggiornamenti. Possiede un gran numero di metodi che permettono di svolgere tutte le operazioni necessarie. Il primo passo è ottenere una istanza della classe, attraverso il seguente metodo:

- **`static UpdateManager getInstance()`**

Altri metodi verranno mostrati in seguito.

3.4 Configurare la policy

La policy è rappresentata attraverso un oggetto di tipo **`UpdatePolicy`**, ottenuto mediante la classe ausiliaria **`UpdatePolicy.Builder`**, che permette di specificarne i parametri attraverso i suoi metodi:

- **`UpdatePolicy.Builder setPolicy(int policy_strategy)`**
  Determina quali step verranno eseguiti automaticamente dal servizio di aggiornamento.

Si possono passare solo le seguenti tre costanti definite nella classe **UpdatePolicy**:

- **POLICY_CHECKS_ONLY**
  Viene effettuato solamente il controllo della presenza o meno di nuovi aggiornamenti. L'installazione e il riavvio del dispositivo devono essere gestiti via codice.
- **POLICY_APPLY_ONLY**
  Oltre a controllare la presenza di nuovi aggiornamenti, questa policy gestisce anche il download e l'installazione dell'aggiornamento. Tuttavia il riavvio del dispositivo deve essere gestito dal programmatore.
- **POLICY_APPLY_AND_REBOOT**
  È l'impostazione di default. Si occupa di tutti gli step in automatico, dal controllo all'installazione dell'aggiornamento e il successivo riavvio.

- **UpdatePolicy.Builder setApplyDeadline (long numero, TimeUnit unità)**
  Imposta la deadline a numero di unità. Il timeout parte quando al dispositivo viene notificata la presenza di un nuovo aggiornamento. Al termine della deadline l'aggiornamento automaticamente procede con il download e installazione, e riavvio del dispositivo. Qualora il riavvio del dispositivo risulti essere una operazione critica, è possibile impostare la deadline ad un valore molto alto; così facendo tutto il processo di update deve essere gestito via codice. Per chiarire meglio l'uso di questo metodo, in questo esempio la deadline viene impostata a ventiquattro ore:
  `builder.setApplyDeadline(24, TimeUnit.HOURS);`
- **UpdatePolicy.Builder build()**
  Restituisce un oggetto di tipo **UpdatePolicy** con i parametri appena impostati.

La policy poi deve essere assegnata al dispositivo. Per far ciò si usa il metodo:

- **void setPolicy(UpdatePolicy policy)**
  Sull'istanza di **UpdateManager**, passando come parametro l'oggetto di tipo **UpdatePolicy** ottenuto con **build()** precedentemente.

Esempio di impostazione di una policy in cui avviene il controllo degli aggiornamenti e la loro installazione, ma non il riavvio del dispositivo, con deadline impostata a 7 giorni:

```
UpdateManager manager = UpdateManager.getInstance();
UpdatePolicy.Builder builder = new UpdatePolicy.Builder();
```

```
builder.setPolicy(UpdatePolicy.POLICY_APPLY_ONLY);
builder.setApplyDeadline(7, TimeUnit.DAYS);

UpdatePolicy policy = builder.build();
manager.setPolicy(policy);
```

3.5 Configurare il canale di aggiornamento

L'istanza di **`UpdateManager`** permette anche di cambiare il canale di aggiornamento al quale il dispositivo fa riferimento. Il cambio è persistente, ossia viene mantenuto anche dopo il riavvio del dispositivo. Si può operare usando i seguenti metodi:

- **`void setChannel(String canale)`**
  Usato per effettuare il cambio di canale passando come parametro una stringa con il nome del canale scelto. Qualora sia avvenuto un cambio di canale di aggiornamento, al successivo aggiornamento verrà eseguito un factory reset.
- **`String getChannel()`**
  Permette di ottenere una stringa con il nome del canale a cui il dispositivo è associato.

3.6 Controllare la presenza di aggiornamenti via codice

Per controllare la presenza di nuovi aggiornamenti ed eventualmente di procedere con l'installazione, si usa il seguente metodo della classe **`UpdateManager`**:

- **`boolean performUpdateNow(int policy)`**
  La policy passata come parametro sovrascrive temporaneamente la policy impostata attraverso il metodo **`setPolicy(UpdatePolicy policy)`**. Ciò permette, ad esempio, di controllare la presenza di aggiornamenti all'avvio dell'applicazione.
  Il metodo restituisce un valore booleano che indica, quando è falso, che la richiesta non è stata eseguita, ad esempio perché un aggiornamento od un controllo era già in corso.

3.7 Gestire lo stato dell'aggiornamento

Attraverso l'interfaccia **`StatusListener`** è possibile monitorare e gestire lo stato dell'aggiornamento. È possibile registrare e cancellare il listener attraverso due metodi della classe **`UpdateManager`**:

- **`void addStatusListener(StatusListener listener)`**
- **`void removeStatusListener(StatusListener listener)`**

L'interfaccia **`StatusListener`** è provvista di un solo metodo:

- **`void onStatusUpdate(UpdateManagerStatus status)`**
  Viene chiamato ogni volta che lo stato del dispositivo cambia. Il parametro di tipo **`UpdateManagerStatus`** fornisce tutti i valori di interesse.

La classe **`UpdateManagerStatus`**, oltre a definire le costanti che definiscono i vari stati dell'aggiornamento, possiede quattro campi che forniscono tutti i valori relativi al suo stato:

- **`int currentPolicy`**
  La policy attualmente in uso.
- **`int currentState`**
  Lo stato corrente dell'**`UpdateManager`**. È il campo più importante dei quattro.
- **`VersionInfo currentVersionInfo`**
  Memorizza le informazioni relative alla versione di Android Things in uso nel dispositivo.
- **`PendingUpdateInfo pendingUpdateInfo`**
  Memorizza informazioni aggiuntive relative all'aggiornamento in corso.

La classe **`VersionInfo`** salva due stringhe relative alla versione di Android Things e all'ID della build installate, mentre la classe **`PendingUpdateInfo`** fornisce alcuni valori aggiuntivi:

- **`long downloadSize`**
  Dimensione dei file dell'aggiornamento in byte.
- **`float downloadProgress`**
  Percentuale, compresa tra 0.0 e 1.0, relativa al progresso del download dell'aggiornamento.
- **`VersionInfo versionInfo`**
  Informazioni sulla versione di Android Things usata nella build dell'aggiornamento.

3.8 Riavvio del dispositivo

Per riavviare il dispositivo è necessario il seguente metodo della classe **`DeviceManager`**:

- **`void reboot()`**

Questo metodo genera un'eccezione di tipo **`SecurityException`**, se nel manifest dell'applicazione non è incluso il seguente permesso:

```
<uses-permission android:name =
    "com.google.android.things.permission.REBOOT" />
```

Vediamo ora un esempio di controllo e gestione degli aggiornamenti all'avvio dell'applicazione:

```
UpdateManager manager = UpdateManager.getInstance();

StatusListener listener = new StatusListener() {

    @Override
    public void onStatusUpdate(UpdateManagerStatus status) {
        switch(status.currentState) {

            case UpdateStatusManager.STATE_IDLE: {
                Log.v(TAG, "Dispositivo in idle");
                break;
            }

            case UpdateStatusManager.STATE_CHECKING_FOR_UPDATES: {
                Log.v(TAG, "Ricerca di aggiornamenti...");
                break;
            }

            case UpdateStatusManager.STATE_UPDATE_AVAILABLE: {
                long size = status.pendingUpdateInfo.downloadSize;
                long MB = size >> 20;
                Log.v(TAG, "Aggiornamento disponibile: "
                    + MB + " MB");
                break;
            }

            case UpdateStatusManager.STATE_DOWNLOADING_UPDATE: {
                Log.v(TAG, "Download dell'aggiornamento...");
                break;
            }

            case UpdateStatusManager.STATE_FINALIZING_UPDATE: {
                Log.v(TAG, "Installazione dell'aggiornamento...");
                break;
            }
```

```
            case UpdateStatusManager.STATE_UPDATED_NEEDS_REBOOT: {
                Log.v(TAG, "Aggiornamento completato. "
                    + "Riavvio del dispositivo...");

                DeviceManager dm = DeviceManager.getInstance();
                dm.reboot();
                break;
            }

            case UpdateStatusManager.STATE_REPORTING_ERROR: {
                Log.e(TAG, "Errore nell'aggiornamento");
                break;
            }

        }
    }
};

manager.addStatusListener(listener);
manager.performUpdateNow(UpdatePolicy.POLICY_APPLY_ONLY);
```

3.9 Sitografia del capitolo 3

1. Android Things - Guide - Device Update
   https://developer.android.com/things/sdk/apis/update
2. Android Things - Reference - UpdateManager
   https://developer.android.com/reference/com/google/android/things/update/UpdateManager
3. Android Things - Reference - UpdatePolicy
   https://developer.android.com/reference/com/google/android/things/update/UpdatePolicy
4. Android Things - Reference - UpdatePolicy.Builder
   https://developer.android.com/reference/com/google/android/things/update/UpdatePolicy.Builder
5. Android Things - Reference - UpdateManagerStatus
   https://developer.android.com/reference/com/google/android/things/update/UpdateManagerStatus
6. Android Things - Reference - PendingUpdateInfo
   https://developer.android.com/reference/com/google/android/things/update/PendingUpdateInfo
7. Android Things - Reference - VersionInfo
   https://developer.android.com/reference/com/google/android/things/update/VersionInfo

Capitolo 4:
Interagire con le periferiche I/O

4.1 Introduzione

Data la natura della piattaforma Android Things, risultano particolarmente importanti le API che permettono di gestire le periferiche hardware anche a un livello più basso rispetto al classico sistema Android usato negli smartphone, così da permettere l'utilizzo di dispositivi esterni di varia natura. Nel dettaglio, sono disponibili API per gestire i singoli pin GPIO, i singoli pin che permettono la modulazione PWM, i moduli di comunicazione UART, SPI e I2C.

Non sono disponibili API per la gestione di moduli ADC o DAC e, nel caso fosse necessario, starà allo sviluppatore implementare l'hardware apposito per svolgere tali funzionalità.

A prescindere da questa piccola mancanza, le API disponibili permettono di interagire con sensori, attuatori, microcontrollori e moltissimi altri tipi di periferiche. Tutto questo senza grosse limitazioni perché è lo sviluppatore stesso a scegliere cosa implementare nell'hardware e come implementarlo.

Detto così Android Things sembrerebbe anche troppo permissivo rispetto alla filosofia Android che si è sviluppata negli ultimi anni, in realtà, le periferiche possono essere usate solo dopo averle specificate una a una nella creazione della build dalla console.

Oltre a ciò, serve anche specificare il relativo permesso per l'applicazione di usare le periferiche nel manifest:

```
<uses-permission android:name =
    "com.google.android.things.permission.USE_PERIPHERAL_IO" />
```

4.2 Gestione periferiche I/O

Per gestire le varie periferiche I/O disponibili bisogna utilizzare la classe **`PeripheralManager`** la quale mette a disposizione i metodi elencati di seguito:

- **`PeripheralManager getInstance()`**
  Metodo statico che ritorna il riferimento all'istanza di **`PeripheralManager`**.

- **`List<String> getGpioList()`**
  Ritorna la lista dei pin GPIO disponibili.
- **`List<String> getPwmList()`**
  Ritorna la lista dei pin PWM disponibili.
- **`List<String> getUartDeviceList()`**
  Ritorna la lista delle periferiche UART disponibili.
- **`List<String> getSpiBusList()`**
  Ritorna la lista delle periferiche SPI disponibili.
- **`List<String> getI2cBusList()`**
  Ritorna la lista delle periferiche I2C disponibili.
- **`Gpio openGpio(String name)`**
  Prova a ottenere un'istanza dell'oggetto GPIO relativo al pin specificato nella stringa **`name`**. Se tale periferica non è occupata e l'operazione riesce, ritorna il riferimento all'istanza ottenuta, altrimenti lancia un'eccezione **`IOException`**.
- **`Pwm openPwm(String name)`**
  Prova a ottenere un'istanza dell'oggetto PWM relativo al pin specificato nella stringa **`name`**. Se tale periferica non è occupata e l'operazione riesce, ritorna il riferimento all'istanza ottenuta, altrimenti lancia un'eccezione **`IOException`**.
- **`UartDevice openUartDevice(String name)`**
  Prova a ottenere un'istanza dell'oggetto UART relativo alla periferica specificata nella stringa **`name`**. Se tale periferica non è occupata e l'operazione riesce, ritorna il riferimento all'istanza ottenuta, altrimenti lancia un'eccezione **`IOException`**.
- **`SpiDevice openSpiDevice(String name)`**
  Prova a ottenere un'istanza dell'oggetto SPI relativo alla periferica specificata nella stringa **`name`**. Se tale periferica non è occupata e l'operazione riesce, ritorna il riferimento all'istanza ottenuta, altrimenti lancia un'eccezione **`IOException`**.
- **`I2cDevice openI2cDevice(String name, int address)`**
  Prova a ottenere un'istanza dell'oggetto I2C relativo alla periferica specificata nella stringa **`name`** e nel valore di **`address`**. Se tale periferica non è occupata e l'operazione riesce, ritorna il riferimento all'istanza ottenuta, altrimenti lancia un'eccezione **`IOException`**.

Dunque, considerando come funzionano i metodi per ottenere l'accesso alle periferiche, risulta chiaro come sia importante chiudere e rilasciare le periferiche quando si termina di utilizzarle. Per fare ciò, bisogna chiamare il metodo **`close()`** sulla periferica aperta.

Esempio apertura e utilizzo della periferica PWM, procedura simile per tutte le periferiche:

```
PeripheralManager manager = PeripheralManager.getInstance();

try {
    mPwm = manager.openPwm(PWM_NAME);

    mPwm.setPwmFrequencyHz(500);
    mPwm.setPwmDutyCycle(50);
    mPwm.setEnabled(true);

} catch (IOException e) {
    Log.w(TAG, "Unable to access PWM", e);
}
```

4.3 GPIO

L'acronimo GPIO sta per General Purpose Input / Output e, dal punto di vista hardware, non sono altro che i pin del microcontrollore che possono essere programmati per essere usati come input o output digitali. Come suggerisce il nome, possono essere usati per scopi generici, infatti, usando un singolo pin o un gruppo di pin si possono sviluppare anche sistemi complessi. Tipicamente ad ogni singolo GPIO è associato un identificativo numerico.

4.3.1 GPIO: Utilizzo come Output

L'utilizzo di un GPIO come output è piuttosto semplice e, infatti, i metodi disponibili per svolgere questa funzione sono pochi:

- **`void setDirection(int direction)`**
  Imposta il GPIO come input se il parametro passato è **`Gpio.DIRECTION_IN`**.
  Invece, per impostarlo come output si usa **`Gpio.DIRECTION_OUT_INITIALLY_LOW`** o **`Gpio.DIRECTION_OUT_INITIALLY_HIGH`**, che impostano rispettivamente il GPIO inizialmente al livello di tensione basso, ovvero 0V, o alto, tipicamente 3.3V.
- **`void setActiveType(int activeType)`**
  In Android Things usando questo metodo è possibile impostare l'associazione tra i livelli logici e i livelli di tensione fisicamente applicati sui pin dei GPIO. Per fare ciò, si possono usare le constanti **`Gpio.ACTIVE_HIGH`** e **`Gpio.ACTIVE_LOW`** che associano

rispettivamente il livello di tensione alto o basso al livello logico alto.
Questo metodo non modifica il livello di tensione del GPIO, modifica solo le variabili relative all'oggetto.

- **`void setValue(boolean value)`**
  Con questo metodo si imposta il GPIO al livello logico alto passando **`true`** e al livello logico basso passando **`false`**.

Di seguito vediamo un esempio di come è possibile inizializzare un GPIO come output inizialmente al livello di tensione alto e con associazione livello logico alto a livello di tensione basso.
Poi, in un secondo momento, si imposta il valore dell'uscita al livello logico alto, ovvero il livello di tensione basso. È molto importante non confondere livelli di tensione con livelli logici.

```
mGpio.setDirection(Gpio.DIRECTION_OUT_INITIALLY_HIGH);
mGpio.setActiveType(Gpio.ACTIVE_LOW);

...

mGpio.setValue(true);
```

4.3.2 GPIO: Utilizzo come Input (Polling)

Anche l'utilizzo del GPIO come input è piuttosto semplice e i metodi da usare sono quasi gli stessi elencati in precedenza. Ovviamente il metodo per impostare il livello logico del GPIO in questo caso non ha più senso, anzi, ora servirà un metodo per ottenere il livello logico del GPIO:

- **`boolean getValue()`**
  Ritorna il livello logico in cui si trova il pin GPIO.
  Questo metodo funziona anche se il GPIO è impostato come output e ritorna lo stato logico alla quale si è impostata l'uscita. Oltre ad essere di dubbia utilità, nel caso del GPIO come output è da usare con attenzione poiché nell'inizializzazione si imposta un livello di tensione e non un livello logico, ciò può causare confusione.

Nell'esempio seguente vediamo come usare due GPIO, il primo **`mGpio1`** impostato come input e con l'associazione del livello di tensione alto al livello logico alto, il secondo **`mGpio2`** come output inizialmente al livello di tensione alto e con l'associazione del livello di tensione basso al livello logico alto. Poi è illustrato come leggere lo stato di **`mGpio1`** e, se è al livello logico basso, cambiare di stato a **`mGpio2`**.

```
mGpio1.setDirection(Gpio.DIRECTION_IN);
mGpio1.setActiveType(Gpio.ACTIVE_HIGH);
mGpio2.setDirection(Gpio.DIRECTION_OUT_INITIALLY_HIGH);
mGpio2.setActiveType(Gpio.ACTIVE_LOW);

...

if ( mGpio1.getValue() == false ) {
    mGpio2.setValue( ! mGpio2.getValue() );
}
```

4.3.3 GPIO: Utilizzo come Input (Interrupt)

Quando si usa un GPIO come input si può anche scegliere di usarlo come una sorgente di interrupt e, per fare ciò, sono disponibili i seguenti metodi:

- **`void setEdgeTriggerType(int edgeTriggerType)`**
  Serve per impostare quali eventi generano l'interrupt, passando come parametro una delle seguenti costanti:
  - **`Gpio.EDGE_NONE`**: nessun interrupt generato (default).
  - **`Gpio.EDGE_RISING`**: passaggio dal livello logico basso al livello logico alto.
  - **`Gpio.EDGE_FALLING`**: passaggio dal livello logico alto al livello logico basso.
  - **`Gpio.EDGE_BOTH`**: qualsiasi cambio del livello logico sul pin.
- **`void registerGpioCallback`**
  `(Handler handler, GpioCallback callback)`
  Registra una callback per l'evento specificato in precedenza. È possibile passare come primo parametro un oggetto **`Handler`** per eseguire la callback su un altro thread.
- **`void registerGpioCallback(GpioCallback callback)`**
  Fa la stessa cosa del precedente, ma la callback viene eseguita sullo stesso thread.
- **`void unregisterGpioCallback(GpioCallback callback)`**
  Disabilita la callback.

Dunque, per programmare le operazioni da compire nella callback, non ci resta che implementare l'interfaccia **`GpioCallback`**, che prevede i seguenti metodi:

- **`boolean onGpioEdge(Gpio gpio)`**
  Dentro questo metodo sarà inserito il codice da eseguire all'occorrenza dell'evento impostato. Ha come parametro il riferimento del GPIO al quale abbiamo agganciato la callback. Al termine di questo metodo il programmatore deve ritornare un valore booleano:

se ritorna **`false`** i successivi eventi saranno ignorati, se invece ritorna **`true`** continueranno ad essere eseguite le callback anche per i successivi eventi.

- **`void onGpioError(Gpio gpio, int error)`**
  Questo metodo invece viene invocato nel caso si verificasse qualche errore nell'accesso alla periferica. In questo caso il programmatore deve assumere che il GPIO non sia più utilizzabile e che non verranno più eseguite le callback.

Vediamo un esempio simile a quanto proposto in precedenza, utilizzando ora la programmazione a interrupt. Più precisamente consideriamo gli stessi **`mGpio1`** e **`mGpio2`** con le stesse configurazioni, ma questa volta associamo una callback al passaggio dal livello logico alto a quello basso sul **`mGpio1`** e al suo interno cambiamo di stato al **`mGpio2`**.

```
mGpio1.setDirection(Gpio.DIRECTION_IN);
mGpio1.setActiveType(Gpio.ACTIVE_HIGH);
mGpio2.setDirection(Gpio.DIRECTION_OUT_INITIALLY_HIGH);
mGpio2.setActiveType(Gpio.ACTIVE_LOW);

mGpio1.setEdgeTriggerType(Gpio.EDGE_FALLING);
mGpio1.registerGpioCallback( new GpioCallback() {
    @Override
    public boolean onGpioEdge(Gpio gpio) {
        try {
            mGpio2.setValue( ! mGpio2.getValue() );
        } catch (IOException e) {
            Log.w(TAG, "Error accessing to mGpio2", e);
        }
        return true;
    }

    @Override
    public void onGpioError(Gpio gpio, int error) {
        Log.w(TAG, "mGpio1: error event " + error);
    }
});
```

4.4 PWM

L'acronimo PWM sta per Pulse Width Modulation ed è un tipo di modulazione digitale che modula il duty-cycle di un'onda quadra a frequenza fissa. Dunque, si potrebbe pensare che un qualsiasi GPIO possa generare un segnale PWM, tuttavia non è così perché se si vuole che il segnale

generato abbia una frequenza e un duty-cycle molto stabili servono dei moduli hardware costituiti da dei contatori che funzionano separatamente dal processore. Più in generale la disponibilità di generare segnali PWM dipende dall'hardware utilizzato.

La particolarità più interessante di un segnale PWM è che si può facilmente demodulare con un filtro passa basso, ottenendo così un segnale analogico che può essere utile per varie applicazioni, soprattutto in assenza di moduli DAC più sofisticati.

Un'altra cosa interessante è la possibilità di sfruttare le stesse API del PWM per generare altri tipi di segnali variando oltre che al duty-cycle anche la frequenza a seconda delle necessità.

Ad esempio, impostando il duty-cycle al 50% e variando la frequenza si possono ottenere dei segnali audio molto semplici. Per produrre segnali audio più sofisticati, invece, si può modulare direttamente il duty-cycle in modo sinusoidale e tenendo la frequenza del PWM sufficientemente alta. In entrambi i casi i segnali andranno filtrati per togliere le componenti ad alta frequenza

A causa di limitazioni fisiche (tempi di apertura e chiusura di switch non nulli) impostando valori di duty-cycle vicini allo 0% o al 100% con una frequenza elevata, si possono verificare delle distorsioni nel segnale generato.

4.4.1 PWM: Utilizzo

Come spiegato, le operazioni eseguibili su un segnale PWM non sono molte, di seguito sono elencati i metodi chiave previsti nell'interfaccia **`Pwm`** per l'utilizzo completo di queste periferiche:

- **`void setPwmFrequencyHz(double freq_hz)`**
  Imposta la frequenza del segnale al valore del parametro espresso in Hertz. Questo metodo agisce direttamente sullo stato del segnale senza dover invocare altri metodi per rendere effettiva la modifica.
- **`void setPwmDutyCycle(double duty_cycle)`**
  Imposta il valore del duty-cycle del segnale al valore del parametro espresso in tra 0 e 100, estremi inclusi. Questo metodo agisce direttamente sullo stato del segnale senza dover invocare altri metodi per rendere effettiva la modifica.
- **`void setEnabled(Boolean enabled)`**
  Abilita o disabilita il segnale PWM. Prima di abilitarlo è necessario impostare la frequenza e il duty-cycle. Si presti attenzione che, se non si disabilita il PWM prima di invocare il metodo **`close()`**, la periferica continuerà a generare il segnale.

Di seguito vediamo un esempio di come è possibile utilizzare una periferica PWM per generare un'onda quadra a 500Hz con un duty-cycle iniziale del 75% e successivamente a 100Hz con una duty-cycle del 25%. Per poi disabilitarla prima di rilasciare la periferica.

```
mPwm.setPwmFrequencyHz(500);
mPwm.setPwmDutyCycle(75);
mPwm.setPwmEnabled(true);

...

mPwm.setPwmFrequencyHz(100);
mPwm.setPwmDutyCycle(25);

...

mPwm.setEnabled(false);
mPwm.close();
```

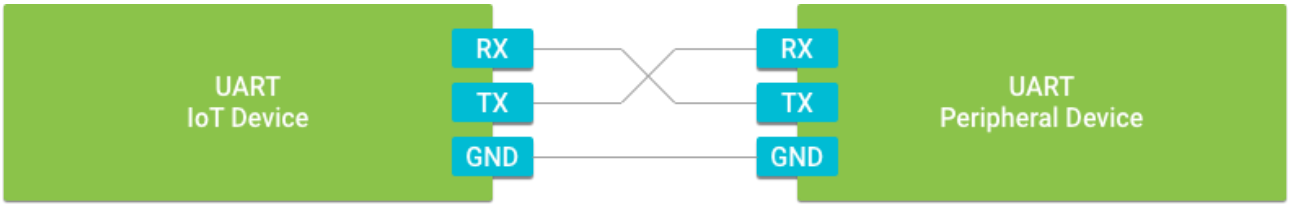
4.5 UART

L'acronimo UART significa Universal Asynchronous Receiver Transmitter ed è un modulo integrato nel microcontrollore che permette una comunicazione seriale asincrona sia in trasmissione che in ricezione. In gergo tecnico a volte le porte UART vengono anche dette porte seriali.

Questo tipo di comunicazione può avvenire solo tra due dispositivi ed essi devono condividere gli stessi parametri di codifica della comunicazione. I parametri essenziali sono:

- Baud rate, ovvero la velocità di trasmissione e ricezione dei dati. Ci sono dei valori standard comunemente usati, ad esempio: 9600, 19200, 38400, 57600, 115200, 230400, etc.
- Bit di dati, ovvero il numero di bit che contengono informazione. Tipicamente 8 bit.
- Bit di parità, ovvero il bit aggiunto in coda ai bit dei dati per eseguire un controllo d'errore. Le impostazioni disponibili per questo parametro sono 3: nessun bit di parità, bit di parità pari oppure bit di parità dispari, cioè che la somma dei bit dei dati con il bit di parità dia un numero pari o un numero dispari.
  È possibile anche impostare il bit di parità come un bit che sia sempre a livello alto (mark) oppure a livello basso (space), in questi casi non avviene il controllo d'errore.
- Bit di stop, ovvero quanti bit di stop devono esserci per segnalare il termine del messaggio. Tipicamente si usano 1 o 2 bit.

Dal punto di vista hardware la connessione UART può essere implementata in vari modi a seconda delle disponibilità e delle esigenze. Per non dilungarci troppo, noi prendiamo in esame solo la modalità senza hardware flow control e modem control: che usa solo due linee per lo scambio di dati in modo seriale, ciascuna simplex. Oltre a queste due linee è ovviamente necessaria la condivisione del potenziale di riferimento (GND).



4.5.1 UART: Configurazione parametri

La prima cosa da fare quando si intende usare la periferica UART è impostare i parametri della comunicazione e, per fare ciò, sono presenti i seguenti metodi:

- **`void setBaudrate(int rate)`**
  Imposta il baud rate, come spiegato in precedenza, il parametro dovrà essere scelto tra i valori standard tipicamente utilizzati e dovrà essere in accordo con quello usato dall'altro dispositivo con cui si vuole comunicare.
- **`void setDataSize(int size)`**
  Imposta il numero di bit dedicati a contenere l'informazione.
- **`void setParity(int mode)`**
  Imposta la modalità di utilizzo del bit di parità, per fare ciò bisogna passare come parametro una tra le seguenti costanti:
  - **`UartDevice.PARITY_NONE`**: nessun bit di parità.
  - **`UartDevice.PARITY_EVEN`**: bit di parità pari.
  - **`UartDevice.PARITY_ODD`**: bit di parità dispari.
  - **`UartDevice.PARITY_MARK`**: bit di parità sempre a 1.
  - **`UartDevice.PARITY_SPACE`**: bit di parità sempre a 0.
- **`void setStopBits(int bits)`**
  Imposta il numero di bit utilizzati per terminare il messaggio.

4.5.2 UART: Trasmissione e ricezione dati (Polling)

Di seguito sono elencati i metodi utili per gestire in modo basilare la periferica UART:

- **`int write(byte[] buffer, int length)`**
  Riceve come parametri un buffer contenente i dati da inviare e il numero di byte del buffer da inviare. Ritorna un intero corrispondente al numero di byte che ha inviato.
- **`int read(byte[] buffer, int length)`**
  Riceve come parametri un buffer dove inserirà i dati letti e il numero di byte da leggere. Ritorna un intero corrispondente al numero di byte che ha letto. Ritorna zero se non c'era alcun byte da leggere. Per assicurarsi di aver letto tutti i byte disponibili bisogna invocare questo metodo ripetutamente finché il numero di byte letti non è uguale a zero.
- **`void flush(int direction)`**
  Questo metodo serve per pulire i registri della periferica UART usati per tenere temporaneamente i dati in ingresso e in uscita. Passando come parametro una tra le seguenti costanti è possibile scegliere su quale buffer agire:
  - **`UartDevice.FLUSH_IN`**: buffer d'ingresso (dati ricevuti ma non letti).
  - **`UartDevice.FLUSH_OUT`**: buffer d'uscita (dati scritti ma non trasmessi).
  - **`UartDevice.FLUSH_IN_OUT`**: entrambi i buffer.

4.5.3 UART: Ricezione dati (Interrupt)

Considerata la natura asincrona, risulta fondamentale poter gestire la periferica UART tramite interrupt. Proprio per questo motivo sono disponibili i seguenti metodi:

- **`void registerUartDeviceCallback (Handler handler, UartDeviceCallback callback)`**
  Registra una callback da invocare quando saranno ricevuti nuovi dati. È possibile passare come primo parametro un oggetto **`Handler`** per eseguire la callback su un altro thread.
- **`void registerUartDeviceCallback(UartDeviceCallback callback)`**
  Fa la stessa cosa del precedente, ma la callback viene eseguita sullo stesso thread.
- **`void unregisterUartDeviceCallback (UartDeviceCallback callback)`**
  Disabilita la callback.

Dunque, per programmare le operazioni da compire nella callback, non ci resta che implementare l'interfaccia **`UartDeviceCallback`**, che prevede i seguenti metodi:

- **`boolean onUartDeviceDataAvailable(UartDevice uart)`**
  Dentro questo metodo sarà inserito il codice da eseguire. Ha come parametro il riferimento della UART alla quale abbiamo agganciato la callback. Al termine di questo metodo il programmatore deve ritornare un valore booleano: se ritorna **`false`** i successivi eventi saranno ignorati (equivale a invocare **`unregisterUartDeviceCallback`**), se invece ritorna **`true`** continueranno ad essere eseguite le callback anche per i successivi eventi.
- **`void onUartDeviceError(UartDevice uart, int error)`**
  Questo metodo invece viene invocato nel caso si verificasse qualche errore nell'accesso alla periferica. In questo caso il programmatore deve assumere che la UART non sia più utilizzabile e che non verranno più eseguite le callback.

Nella parte iniziale dell'esempio seguente vediamo come inizializzare la periferica UART usando parametri di comunicazione: baud rate = 9600, numero di bit = 8, nessun bit di parità e un bit di stop. Dopo l'inizializzazione, viene fatto un reset dei registri d'ingresso e d'uscita, così da eliminare eventuali residui di utilizzi precedenti.

```
mUart.setBaudrate(9600);
mUart.setDataSize(8);
mUart.setParity(UartDevice.PARITY_NONE);
mUart.setStopBits(1);

mUart.flush(UartDevice.FLUSH_IN_OUT);
```

Nella seconda parte dell'esempio vediamo come agganciare una callback da eseguire quando verranno ricevuti dei dati. Si noti che il metodo onUartDeviceDataAvailable ritorna false e dunque la callback sarà eseguita solo una volta e poi sarà disabilitata.

All'interno della callback, nel metodo onUartDeviceDataAvailable, è stato inserito del codice che legge tutti i dati disponibili e poi li ritrasmette tali e quali.

```
mUart.registerUartDeviceCallback( new UartDeviceCallback() {

@Override
    public boolean onUartDeviceDataAvailable(UartDevice uart) {
        byte[] buffer = new byte[20];
        int count;
        try {
            while ((count = uart.read(buffer, buffer.length))>0) {
                uart.write(buffer, count);
            }
```

```
        } catch (IOException e) {
            Log.w(TAG, "UART exception", e);
        }

        return false;
    }

    @Override
    public void onUartDeviceError(UartDevice uart, int error) {
        Log.w(TAG, "UART error event " + error);
    }
});
```

4.6 SPI

L'acronimo SPI significa Seria Peripheral Interface ed è un'altra interfaccia di comunicazione seriale che, a differenza della UART, permette di comunicare con un numero illimitato di dispositivi. Infatti, la struttura della rete prevede un dispositivo master che controlla tutti gli altri dispositivi slave.

Un'altra caratteristica importante della SPI è che implementa una comunicazione sincrona e, dunque, la sua velocità di trasmissione può arrivare a valori più alti rispetto a quelli della UART, che è asincrona. La particolarità più curiosa della comunicazione SPI è che l'invio e la ricezione dei dati avvengono esattamente nello stesso momento ed è sempre il master a decidere quando farlo.

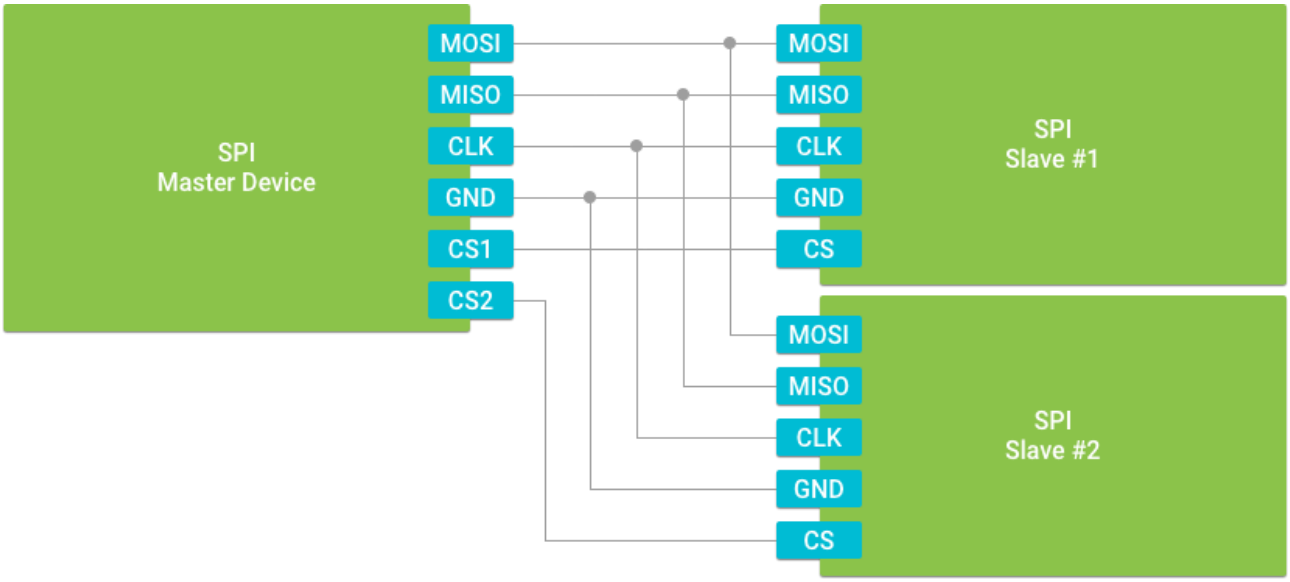
Anche per l'interfaccia SPI è possibile impostare alcuni parametri della comunicazione:

- Modalità di sincronizzazione sul clock, ovvero in corrispondenza a quali eventi sul clock effettuare l'operazione di trasmissione e l'operazione di lettura dei dati. Senza entrare troppo nel dettaglio, è bene sapere che esistono quattro modalità standard.
- Frequenza del clock, cioè la frequenza del segnale di sincronizzazione della comunicazione.
- Bit per parola, ovvero il numero di bit che contengono informazione.
- Ordinamento dei bit, cioè se sono trasmessi prima i bit meno significativi o prima quelli più significativi.

Dal punto di vista hardware le connessioni sono quattro per ogni dispositivo slave, più il potenziale della massa. Risultano particolarmente interessanti: la linea MOSI (master output slave input), che

trasferisce i dati dal master agli slave, la linea MISO (master input slave output), che trasferisce i dati dagli slave al master, e la linea CLK che è il segnale di clock.

Inoltre, come si vede nella figura seguente, ci sono le linee CS (chip select) che sono separate per ogni slave. Queste hanno lo scopo di abilitare il singolo slave con il quale il master desidera comunicare, mentre gli altri slave saranno disabilitati.



4.6.1 SPI: Configurazione parametri

Di seguito sono elencati i metodi disponibili per configurare la periferica SPI (master) relativa al singolo slave connesso sul bus specificato nel metodo **`openSpiDevice`**:

- **`void setMode(int mode)`**
  Imposta la modalità di sincronizzazione sul segnale di clock. Come parametro può ricevere la costante relativa alla modalità desiderata: **`SpiDevice.MODE0`**, **`SpiDevice.MODE1`**, **`SpiDevice.MODE2`** o **`SpiDevice.MODE3`**.
- **`void setFrequency(int frequencyHz)`**
  Imposta la frequenza del clock prendendo come parametro il valore espresso in Hz.
- **`void setBitsPerWord(int bitsPerWord)`**
  Imposta il numero di bit contenenti informazione.
- **`void setBitJustification(int justification)`**
  Imposta l'ordine dei bit MSB o LSB prendendo come parametro una tra le due costanti:
  - **`SpiDevice.BIT_JUSTIFICATION_MSB_FIRST`**
  - **`SpiDevice.BIT_JUSTIFICATION_LSB_FIRST`**

4.6.2 SPI: Trasferimento dati (full-duplex)

Per il trasferimento dei dati c'è un unico metodo che permette di fare tutto contemporaneamente:

- **`void transfer(byte[] txBuffer, byte[] rxBuffer, int length)`**
  Trasmette i dati contenuti nell'array **`txBuffer`** e salva i dati ricevuti nell'array **`rxBuffer`**, come terzo parametro prende il numero di byte che deve trasferire.

Nell'esempio seguente viene mostrato come inizializzare la periferica SPI e, in un secondo momento, come inviare 10 byte e riceverne altri 10 nello stesso momento:

```
mSpi.setMode(SpiDevice.MODE0);
mSpi.setFrequency(16000);
mSpi.setBitsPerWord(8);
mSpi.setBitJustification(SpiDevice.BIT_JUSTIFICATION_MSB_FIRST);

...

byte[] write = new byte[] {7, 44, 9, 37, 90, 72, -20, 8, 77, 120};
byte[] read = new byte[write.length];
mSpi.transfer(write, read, write.length);
Log.i(TAG, "Byte letti: " + Arrays.toString(read));
```

4.6.3 SPI: Trasmissione e ricezione dati (half-duplex)

Come spiegato, la trasmissione e la ricezione dei dati avviene contemporaneamente. Tuttavia, è possibile gestire la comunicazione anche ignorando ciò che si è ricevuto finché si stava scrivendo e trasmettere una serie di 0 finché si sta leggendo. Proprio a questo servono i seguenti metodi:

- **`void write(byte[] buffer, int length)`**
- **`void read(byte[] buffer, int length)`**

Non sono altro che una semplificazione del metodo **`transfer`** visto in precedenza.

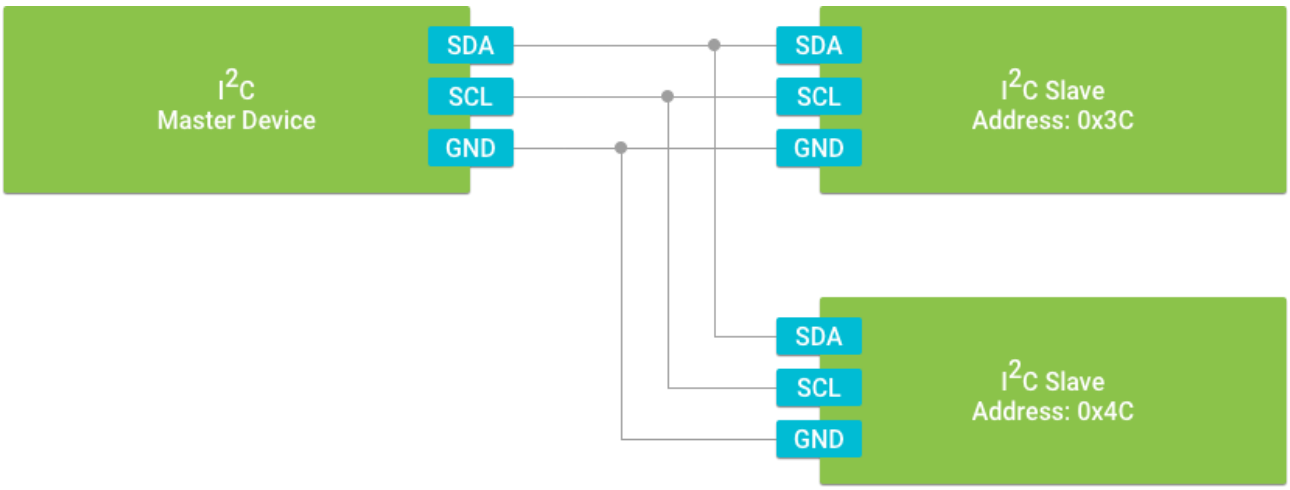
4.7 I2C

L'acronimo I2C (o anche I2C o IIC) significa Inter Integrated Circuit ed è un'interfaccia di comunicazione seriale che permette di far interagire fino a 128 dispositivi sulla stessa linea di comunicazione. Nella struttura della rete sono previsti dispositivi slave e un master, come la SPI.

La I2C è una comunicazione sincrona, ma la sua massima velocità di trasmissione non è particolarmente elevata ed è impostabile tra alcune frequenze standard.

Contrariamente alla SPI, il dispositivo master identifica gli slave tramite il loro indirizzo che generalmente a 7 bit e deve essere univoco nella rete. Siccome lo scambio di dati prevede dei meccanismi di sincronizzazione piuttosto complessi che vengono fatti automaticamente, vedremo solo in modo superficiale come funziona la comunicazione. Essenzialmente il master può inviare una richiesta di un numero definito di byte relativi a un registro di uno slave indentificato dal suo indirizzo, appena completata la richiesta lo slave inizierà a rispondere. Il master può anche dare dei comandi allo slave scrivendo sui suoi registri.

Dal punto di vista hardware le linee sono solo due, più il potenziale della massa, e sono condivise tra tutti gli elementi della rete. Nel dettaglio, la linea SDA è una linea half-duplex dove passano i dati scambiati tra i dispositivi, mentre la linea SCL è il segnale di clock emesso dal master.



4.7.1 I2C: Utilizzo

In Android Things l'interfaccia **`I2cDevice`** attualmente disponibile è riferita a un singolo slave e non lascia spazio a molte impostazioni. Infatti, è molto semplificata e permette solo di leggere i valori dei registri dello slave, oppure di scrivere sui registri disponibili dello slave.

Metodi per leggere dai registri:

- **`void readRegBuffer(int reg, byte[] buffer, int length)`**
  Legge i valori dei registri partendo dall'indirizzo specificato nel primo parametro reg e proseguendo per i successivi registri, per un totale corrispondente al terzo parametro **`length`** (al massimo 32). Salva i valori letti sul **`buffer`** passato come secondo parametro.

- **`byte readRegByte(int reg)`**
  Ritorna il byte letto dal registro specificato nel parametro **`reg`**.
- **`short readRegWord(int reg)`**
  Ritorna uno short contenente i valori letti dal registro specificato nel parametro **`reg`** e dal successivo. Il valore del primo registro viene considerato come byte meno significativo.

Metodi per scrivere nei registri:

- **`void writeRegBuffer(int reg, byte[] buffer, int length)`**
  Scrive i valori contenuti nel **`buffer`** nei registri partendo dall'indirizzo specificato nel primo parametro **`reg`** e proseguendo per i successivi registri, per un totale corrispondente al terzo parametro **`length`** (al massimo 32).
- **`void writeRegByte(int reg, byte data)`**
  Scrive un byte passato nel parametro data nel registro specificato nel parametro **`reg`**.
- **`void writeRegWord(int reg, short data)`**
  Scrive uno short passato nel parametro **`data`** nel registro specificato nel parametro **`reg`** e nel successivo. Il valore del primo registro viene considerato come byte meno significativo.

Sono disponibili anche dei metodi per una comunicazione raw, cioè per lo scambio di dati senza l'utilizzo di registri:

- **`void read(byte[] buffer, int length)`**
  Legge un numero di byte specificati nel parametro **`length`** e li salva nel **`buffer`**.
- **`void write(byte[] buffer, int length)`**
  Scrive un numero di byte specificati nel parametro **`length`** e contenuti nel **`buffer`**.

4.8 Sitografia del capitolo 4

1. Android Things - Guide - Peripheral I/O
   https://developer.android.com/things/sdk/pio/
2. Android Things - Reference - com.google.android.things.pio
   https://developer.android.com/reference/com/google/android/things/pio/package-summary

3. Android Things - Reference - PeripheralManager
   https://developer.android.com/reference/com/google/android/things/pio/PeripheralManager

4. Android Things - Guide - GPIO
   https://developer.android.com/things/sdk/pio/gpio
5. Wikipedia - General purpose input/output
   https://en.wikipedia.org/wiki/General-purpose_input/output
6. Android Things - Reference - Gpio
   https://developer.android.com/reference/com/google/android/things/pio/Gpio
7. Android Things - GpioCallBack
   https://developer.android.com/reference/com/google/android/things/pio/GpioCallback

8. Android Things - Guide - PWM
   https://developer.android.com/things/sdk/pio/pwm
9. Wikipedia - Pulse-width modulation
   https://en.wikipedia.org/wiki/Pulse-width_modulation
10. Android Things - Reference - Pwm
    https://developer.android.com/reference/com/google/android/things/pio/Pwm

11. Android Things - Guide - UART
    https://developer.android.com/things/sdk/pio/uart
12. Wikipedia - Universal asynchronous receiver-transmitter
    https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
13. Android Things - Reference - UartDevice
    https://developer.android.com/reference/com/google/android/things/pio/UartDevice
14. Android Things - Reference - UartDeviceCallback
    https://developer.android.com/reference/com/google/android/things/pio/UartDeviceCallback

15. Android Things - Guide - SPI
    https://developer.android.com/things/sdk/pio/spi
16. Wikipedia - Serial Peripheral Interface Bus
    https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
17. Android Things - Reference - SpiDevice
    https://developer.android.com/reference/com/google/android/things/pio/SpiDevice

18. Android Things - Guide - I2C
    https://developer.android.com/things/sdk/pio/i2c
19. Wikipedia - I²C
    https://en.wikipedia.org/wiki/I%C2%B2C
20. Android Things - Reference - I2cDevice
    https://developer.android.com/reference/com/google/android/things/pio/I2cDevice

21. Google's IoT Developers Community
    https://plus.google.com/communities/107507328426910012281

Capitolo 5:
Il nostro progetto Distance Alert

5.1 Scopo del progetto

Il progetto Distance Alert nasce con lo scopo di realizzare una applicazione che esemplificasse le funzionalità specifiche di Android Things, in particolare le API di gestione delle periferiche I/O.

La nostra applicazione permette di monitorare la distanza tra la Raspberry Pi e un oggetto mobile (tag), scelto tra quelli disponibili nelle vicinanze. È stato previsto un allarme, composto da un LED e da un buzzer, il quale scatta al superamento di un valore di soglia, impostabile nell'applicazione.

Per la misura della distanza abbiamo usato delle schede di sviluppo Decawave che svolgono questo compito in modo autonomo e con le quali è possibile scegliere se comunicare via UART o SPI.

L'interfaccia utente dell'applicazione è estremamente semplice: a sinistra vi è una lista dei tag disponibili, da cui è possibile selezionare l'oggetto di cui monitorare la distanza visualizzata al centro. A destra vengono mostrati la distanza di soglia dell'allarme, variabile attraverso i bottoni laterali, e uno switch per variare il bus di comunicazione con la scheda Decawave, tra SPI e UART.

La funzionalità di cambio bus di comunicazione è stata implementata a puro scopo esemplificativo, in una possibile applicazione futura lo sviluppatore potrà scegliere definitivamente un bus o l'altro a seconda delle esigenze.

5.2 Descrizione Software

Sulla Raspberry Pi 3 B usata per il progetto è stata montata un microSD contenente una build che usa la versione 1.0 di Android Things e che ha come periferiche abilitate:

- **SPI**: SPI0.
- **UART**: MINIUART.
- **GPIO**: BCM16 e BCM26 (con resistenza di pull-down).
- **PWM**: PWM0/PWM1.

Per quanto riguarda il software dell'applicazione, essenzialmente è stato realizzato il pacchetto **`group107.distancealert`** il quale comprende:

- **`DriverDWM.java`**: driver generale che si occupa di gestire la comunicazione con il modulo, sia tramite SPI che tramite UART.
- **`DistanceController.java`**: classe che utilizza **`DriverDWM`** per fornire funzionalità specifiche per l'app. In particolare, si occupa di determinare quali oggetti mobili si siano appena connessi e disconnessi, oltre a fornire valori aggiornati della posizione.
- **`AllTagsListener.java`** e **`TagListener.java`**: interfacce utili per gestire in maniera asincrona i dati ricevuti da **`DriverDWM.java`**, con la differenza che la prima è dedicata a tutti i tag, mentre la seconda ad un singolo tag specifico.
- **`DistanceAlarm.java`**: classe già specializzata per le necessità dell'applicazione, ha lo scopo di gestire i dispositivi di allarme tramite **`PheripheralManager`**, **`Gpio`** e **`Pwm`**;
- **`MainActivity.java`**: activity principale dell'applicazione, questa classe sfrutta tutte le altri classi e gestisce l'interfaccia utente.
- **`SleepHelper.java`**: classe ausiliaria che implementa diversi metodi di sleep per i thread. Questa classe ha lo scopo di conferire maggiore leggibilità al codice.

Ogni classe è stata dotata di molti commenti, al fine di documentare al meglio le varie sezioni di codice. Inoltre, sono presenti svariati log, divisi per TAG specifici, usati per il debug dell'applicazione.

Nel metodo **`onCreate()`** di **`MainActivity.java`** vengono inizializzati gli elementi grafici dell'activity, oltre a instanziare un oggetto di **`DistanceController`**. Quest'ultimo si appoggia a **`DriverDWM`** per gestire la comunicazione, decodifica i dati ricevuti dal driver e li fornisce alla **`MainActivity`** attraverso un sistema di listener. In particolare, la **`MainActivity`** sfrutta **`AllTagsListener`** per il monitoraggio della lista degli ID e **`TagListener`** per mostrare i dati di un ID specifico.

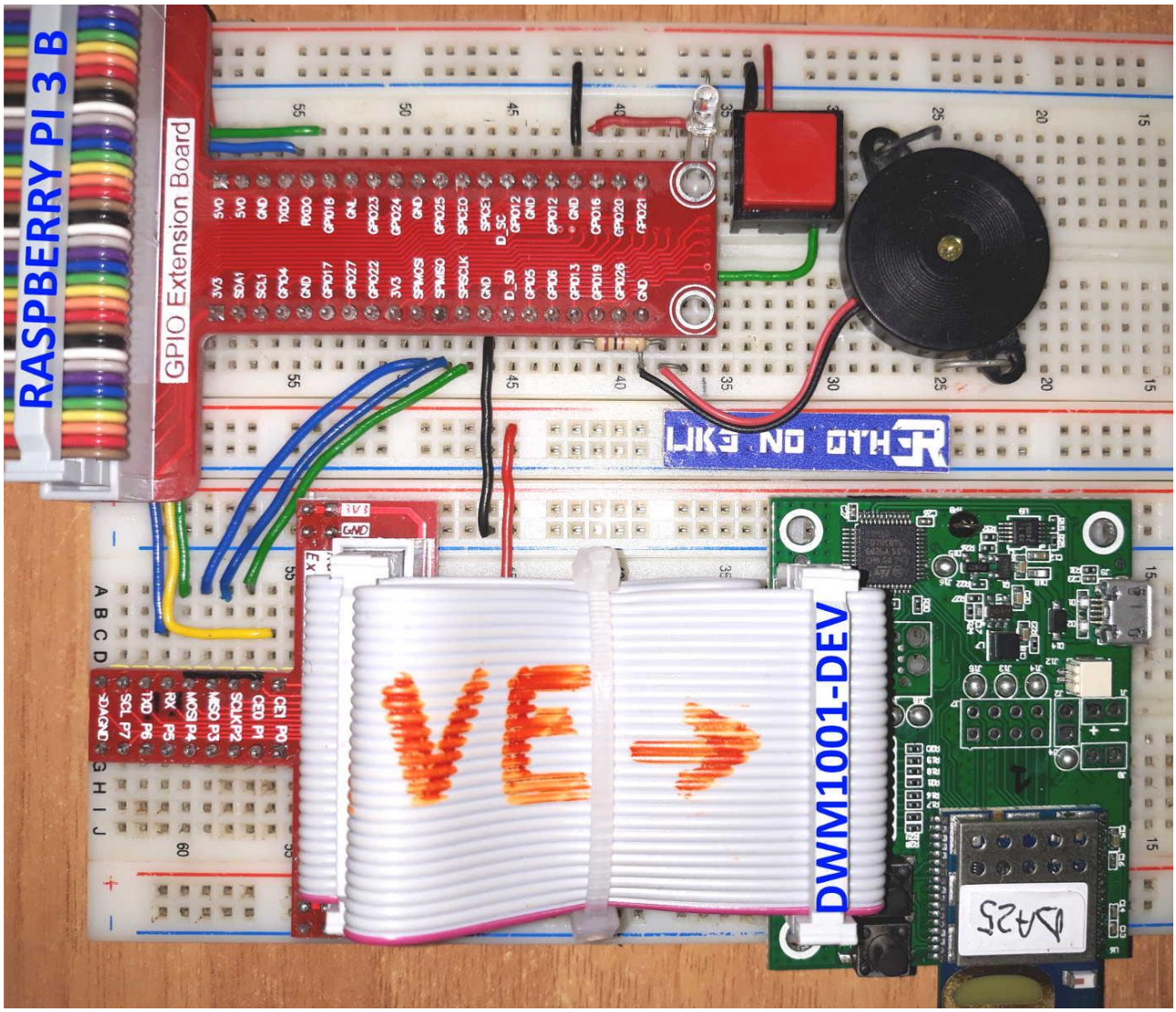
5.3 Descrizione Hardware

Nel progetto abbiamo collegato alla Raspberry Pi 3 B:

- Uno schermo tramite l'uscita HDMI.

- Una tastiera e un mouse tramite USB.
- Un pulsante al GPIO26, con il contatto normalmente aperto a VCC e il contatto normalmente chiuso a GND.
- Un LED blu con una resistenza in serie da 220Ω tra il GPIO16 e GND.
- Un buzzer con una resistenza in serie da 100Ω tra il GPIO13 (PWM1) e GND.
- Un modulo DWM1001-DEV collegato via UART tramite i soli pin Tx e Rx.
- Lo stesso modulo DWM1001-DEV collegato anche via SPI collegato via SPI.

Di seguito una foto del circuito montato su breadboard. Il collegamento con la Raspberry Pi viene effettuato usando un bus da 40 pin che permette di espandere i contatti comodamente sulla breadboard. La stessa cosa viene effettuata per il modulo DWM con un bus da 26 pin.

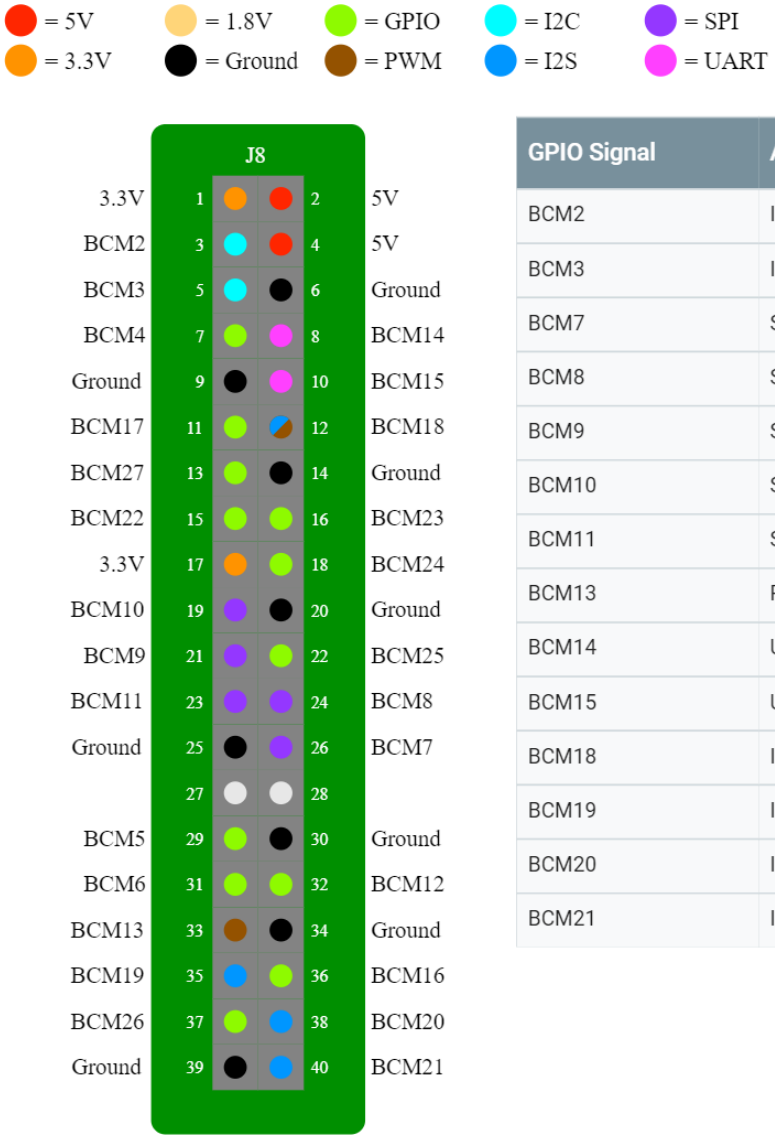


5.3.1 Raspberry Pi 3 B

La scheda Raspberry Pi 3 B ha un buon numero di periferiche di input / output disponibili.

Di seguito elenchiamo le principali usate nel progetto, illustrando la relativa codifica da usare nel codice dell'app per Android Things:

- Una ventina di GPIO identificati come **`"BCMx"`** dove la **`x`** sarà il numero del BCM scelto. In generale, tutti i pin possono essere usati come GPIO, ma alcuni di essi possono svolgere anche altre funzioni, come si può vedere nell'immagine alla fine di questo paragrafo.
- 2 pin PWM che sono indentificati come **`"PWM0"`** sul BCM18 e **`"PWM1"`** sul BCM13.
- Un'interfaccia UART identificata come **`"MINIUART"`** e posizionata sui BCM14 e BCM15.
- Un'interfaccia SPI con la possibilità di collegare 2 dispositivi slave identificati come **`"SPI0.0"`**, che usa come chip select il BCM8, e **`"SPI0.1"`**, che invece usa il BCM7. I pin usati per i segnali MISO, MOSI e SCK sono i BCM8, BCM9 e BCM10.



GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	

5.3.2 Decawave DWM1001-DEV

Le board DWM1001-DEV sono delle schede di sviluppo usate per testare i moduli DWM1001. Questi moduli si connettono automaticamente ad altri moduli dello stesso tipo inviando segnali UWB a bassa potenza, poi sfruttando l'algoritmo two-way-ranging calcolano la distanza dai vari nodi della rete. Essendo questa tecnologia abbastanza nuova, essi sono tuttora in fase di sviluppo e possono avere dei malfunzionamenti, ad esempio, abbiamo notato che a volte il modulo non risponde per qualche minuto. Tuttavia, sono molto interessanti e sicuramente, una volta ottimizzati, avranno potenzialmente molti ambiti d'utilizzo.

I produttori hanno messo a disposizione molte API utilizzabili in modo molto simile sia via UART che SPI. Infatti, essenzialmente con il modulo si comunica usando dei pacchetti codificati con il formato TLV (tag-length-value). Di fatto noi nel nostro progetto usiamo solo due tipi di richieste:

- **Get Update Rate** (dwm_upd_rate_get)
  Questa API restituisce la frequenza di aggiornamento della posizione del modulo. Nel progetto è usata esclusivamente per verificare che la connessione con modulo funzioni. Di fatto è stata scelta perché prevede dei messaggi corti e perché non altera lo stato del modulo.
  Per fare questa richiesta al modulo è sufficiente inviare un pacchetto che ha come tag **`0x04`** e con length **`0x00`**, cioè senza alcun value. Vediamo un esempio di richiesta e risposta:

Example:

TLV request	
Type	**Length**
0x04	0x00

TLV response						
Type	**Length**	**Value**	**Type**	**Length**	**Value**	
		err_code			**update_rate**, first 2 bytes, indicating the position update interval in multiples of 100 ms. E.g. 0x0A 0x00 means 1.0s interval.	**update_rate_stationary**, second 2 bytes, indicating the stationary position update interval in multiples of 100 ms. E.g. 0x32 0x00 means 5.0s interval.
0x40	0x01	0x00	0x45	0x04	0x0A 0x00	0x32 0x00

- **`Get Location`** (dwm_loc_get)
  Questa API restituisce le coordinate del modulo nello spazio (nel caso la rete lo preveda) e poi tutte le distanze in mm da ogni modulo (identificato con il relativo ID) connesso alla rete. Nel caso un modulo non risulti più connesso, conserva comunque l'ultima distanza rilevata senza rimuoverlo dall'elenco. Nel nostro progetto è usata per rilevare l'elenco dei moduli connessi e le relative distanze da monitorare.
  Per fare questa richiesta al modulo è sufficiente inviare un pacchetto che ha come tag **`0x0C`**

e con length `0x00`, cioè senza alcun value. Vediamo un esempio di richiesta e risposta:

Example 1 (Tag node):

TLV request	
Type	**Length**
0x0C	0x00

TLV response										
Type	**Length**	**Value**	**Type**	**Length**	**Value**				**Type**	**Length**	**Value**
		err_code			**position**, 13 bytes						1 byte, number of distances encoded in the value
					4-byte x	4-byte y	4-byte z	1-byte quality factor			
0x40	0x01	0x00	0x41	0x0D	0x4b 0x0a 0x00 0x00 0x1f 0x04 0x00 0x00 0x9c 0x0e 0x00 0x00 0x64				0x49	0x51	0x04

TLV response (residue of the frame from previous table)								
Value								
2 bytes UWB address	4-byte distance	1-byte distance quality factor	position in standard 13 byte format	…	2 bytes UWB address	4-byte distance	1-byte distance quality factor	position in standard 13 byte format
position and distance AN1				AN2, AN3	position and distance AN4			

5.4 Comunicazione tra Raspberry Pi 3 B e DWM1001-DEV

Come spiegato in precedenza, nel nostro progetto il modulo DWM1001-DEV è collegato sia via UART che SPI per sperimentare entrambi i metodi di comunicazione. I pacchetti scambiati sono identici, ma la differenza sostanziale sta nella gestione della comunicazione poiché la UART è asincrona, mentre la SPI è sincrona.

5.4.1 Comunicazione via UART

La comunicazione UART tra Raspberry e il modulo DWM è concettualmente molto semplice: la Raspberry invia la richiesta e poi il modulo invia la risposta. Tuttavia, la natura asincrona della comunicazione UART impone che la Raspberry non possa aspettare un tempo indeterminato. Per questo, nel nostro progetto abbiamo previsto che se la risposta non arriva entro 30ms supponiamo che quella comunicazione non sia andata a buon fine e, dunque, viene lanciata un'eccezione. Lo stesso succede se il modulo risponde in modo errato.

5.4.2 Comunicazione via SPI

La comunicazione SPI tra Raspberry e il modulo DWM avviene essenzialmente in 3 fasi:

- Invio richiesta: la Raspberry invia il pacchetto TLV contenente la richiesta e per ogni byte inviato, viene ricevuto **`0xff`**.
- Attesa risposta: in questa fase il modulo deve elaborare la richiesta e preparare la risposta. La Raspberry deve inviare un byte **`0xff`** alla volta, finché non riceve una risposta diversa da **`0x00`**, tale risposta corrisponderà alla lunghezza della risposta che la Raspberry dovrà ricevere dal modulo DWM.
- Ricezione risposta: la Raspberry invia un numero di byte **`0xff`** corrispondenti alla lunghezza indicata dal modulo e, contemporaneamente, riceve i byte di risposta.

Questa procedura è necessaria poiché il modulo DWM è uno slave nella comunicazione SPI, dunque non può comunicare niente al master se non è il master stesso a ordinarlo.

Come per la UART, anche per la SPI abbiamo imposto un tempo limite di attesa della risposta che, in questo caso, è di 10ms.

5.5 Sitografia del capitolo 5

1. Android Things - Guide - Raspberry Pi I/O
   https://developer.android.com/things/hardware/raspberrypi-io
2. Decawave - DWM1001-DEV
   https://www.decawave.com/products/dwm1001-dev
3. Decawave - DWM1001 Firmware Application Programming Interface (Api) Guide
   https://www.decawave.com/sites/default/files/dwm1001-api-guide.pdf
4. Decawave - Datasheet for DWM1001-Dev Board
   https://www.decawave.com/sites/default/files/dwm1001-dev_datasheet.pdf

---