Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Sistema de monitorización de señales basado en IOIO y Android

Autor: Javier Soriano Ruiz

Tutor: Antonio Luque Estepa

Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Sistema de monitorización de señales basado en IOIO y Android

Autor:
Javier Soriano Ruiz

Tutor:
Antonio Luque Estepa
Profesor titular

Dep. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Grado: Sistema de monitorización de señales basado en IOIO y Android

Autor: Javier Soriano Ruiz

Tutor: Antonio Luque Estepa

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis amigos

A mis maestros

Agradecimientos

Durante estos años de vida universitaria ha habido mucha gente importante que ha pasado por mi vida y que me ha ayudado a ir superando poco a poco cada obstáculo nuevo que se presentaba. Por tanto, no puedo dejar de mencionar aquí a mi familia, amigos y profesores.

En primer lugar, tengo que agradecer a mis padres su esfuerzo y dedicación para hacerme llegar hasta aquí, en especial a mi madre, que ha estado siempre apoyándome en los momentos más duros. Mamá, sin ti esto no sería posible. Te quiero.

Los amigos siempre son una burbuja en la que escaparse en esos momentos de agobio, por tanto, no puedo dejar de darles las gracias. Primero, a Cobos, que siempre ha estado ahí para echarme una mano, y segundo, a Adrián Vélez (Bole) por su ayuda con los problemillas que me surgían de programación y a Fran Pérez, que sin su ayuda esta memoria estaría completamente descuadra.

Pero también tengo que darles las gracias a aquellos con los que más que estudio, había "cachondeo", porque gracias a ellos uno se olvidaba un poco de las obligaciones y luego las tomaba con más ganas. Gracias a mis "pimientos", gente de San Diego y telecos (Migue, Dani, Marchal, Richy, Fran, Bole y todos los que han pasado por mi vida estos años).

Finalmente, no puedo dejar de dar las gracias a todos mis maestros, en especial al tutor de este proyecto. Gracias, Antonio, por aguantarme durante todo este tiempo y ayudarme a ver la luz cuando sólo veía oscuridad.

GRACIAS

Javier Soriano Ruiz

Sevilla, 2017

Resumen

El siglo XXI está siendo testigo de la conquista de los microcontroladores. Estos pequeños dispositivos están presentes constantemente en nuestra vida cotidiana (ordenadores, teléfonos, televisión, electrodomésticos, …). Además, su uso es cada vez más extendido aprovechando el "Internet de las cosas" (*IoT*).

Asimismo, a la vez que los microcontroladores han ido evolucionando, cada vez tenemos teléfonos inteligentes más complejos, que nos permiten realizar todo tipo de tareas.

El objetivo de este proyecto es estudiar la integración entre microcontroladores y teléfonos inteligentes (en concreto aquellos con sistema operativo Android). Para ello, se ha utilizado IOIO-OTG.

IOIO es una placa electrónica que proporciona la capacidad de interacción entre el microcontrolador y su hardware externo con dispositivos Android, empleando para ello un protocolo propio para la comunicación entre IOIO y el dispositivo móvil. Además, posee una librería Android que permite la programación del microcontrolador mediante el desarrollo de programas Android.

Este proyecto se centrará en el estudio de IOIO y su integración con Android, donde se planteará una mejora tanto en el Firmware del microcontrolador como en la librería Android que permita el uso de IOIO sin la necesidad de estar conectado al terminal móvil.

Abstract

The 21st century is being witness of the conquests of microcontrollers. These little devices are constantly present in our daily life (computers, smartphones, televisions, electrical household appliances, …). In addition, its use is increasingly extended by taking advantage of the Internet of Things (IoT).

Furthermore, while microcontrollers have been evolving, we have increasingly complex smartphones, which allow us to perform all kinds of tasks.

The aim of this project is to study the integration between microcontrollers and smartphones (specifically those with Android operating system). To this end, IOIO-OTG has been used.

IOIO is an electronic board that provides the ability to interact between the microcontroller and its external hardware with Android devices, using a communication protocol between IOIO and the mobile device. In addition, it has an Android library that allows programming the microcontroller through the development of Android programs.

This project will be focused on the study of IOIO and its integration with Android. There will be an improvement in both the firmware of the microcontroller and the Android library that allows the use of IOIO without the need to be connected to the mobile terminal.

Índice

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Hay personas que luchan un día y son buenas. Hay otras que luchan un año y son mejores. Hay quienes luchan muchos años y son muy buenas. Pero hay las que luchan toda la vida, esas son las imprescindibles

- Bertolt Brecht-

El avance tecnológico del último siglo ha permitido que los sistemas basados en microcontroladores y los terminales inteligentes lleguen a una amplia parte de la población. Esto ha hecho que los *smartphones* sustituyan en muchos casos a algunos accesorios de nuestra vida cotidiana tales como despertadores, cámaras fotográficas e incluso ordenadores.

1.1 Motivación

La demanda de accesorios electrónicos que permitan ampliar las funcionalidades de estos terminales ha aumentado, por lo que resulta de interés encontrar algún tipo de arquitectura que permita añadir nuevas funcionalidades a nuestros dispositivos móviles.

Es aquí donde IOIO juega un papel fundamental para este proyecto. IOIO es un microcontrolador con un firmware que proporciona un protocolo de comunicaciones para dispositivos Android. Mediante este protocolo es posible establecer una comunicación con un terminal Android para programar las diversas funcionalidades del microcontrolador, desde una entrada o salida digital, hasta una lectura de una señal analógica o salida PWM. Así, utilizando la API que se proporciona para ello, es posible programar aplicaciones Android que permitan la interactuación del terminal móvil y el microcontrolador, y ampliar funcionalidades.

Una de las funcionalidades más interesantes que se ha pensado es la de crear un sistema de monitorización de señales, que permita la lectura de una determinada señal dada por un sensor, almacenarla en memoria y posteriormente enviarla al teléfono para analizarla.

Sin embargo, el desarrollo de este sistema conllevaría una serie de dificultades, puesto que, en un principio, ni el firmware del microcontrolador, ni la librería, proporcionarían directamente esta funcionalidad.

Sería, por tanto, interesante, llevar a cabo unas modificaciones que permitan la ejecución de este proceso para dotar de una nueva funcionalidad a IOIO y al terminal móvil.

1.2 Objetivos

Se va a desarrollar una solución que permita monitorizar una señal analógica de entrada a IOIO durante un determinado tiempo. Para ello, se creará una aplicación Android con las siguientes características:

- La aplicación debe ser capaz de programar el comienzo del muestreo de la señal analógica. Para ello, constará de un botón que dará la orden.
- Una vez que comience a muestrearse la señal, el dispositivo Android podrá ser desconectado del microcontrolador hasta que el usuario desee conocer la evolución de la señal de entrada.
- Para terminar el muestreo de la señal, la aplicación dispondrá de un botón que permita el fin de este y la obtención de las muestras en el terminal.
- La aplicación tendrá otro botón, que redirigirá a una gráfica que mostrará la evolución de la señal durante el tiempo de muestreo.
- Las muestras se podrán guardar en un fichero con formato CSV que se almacenará en la memoria interna del dispositivo y que tendrá como nombre la fecha y hora en la que comenzó el muestreo.

Para cumplir todos estos requisitos será necesario realizar una serie de modificaciones en el firmware de IOIO y en la librería de Android.

Los cambios a realizar en el firmware deberán ser los siguientes:

- Modificación del protocolo de comunicación para satisfacer las nuevas necesidades.
- Programación de la conversión analógico-digital de la señal de entrada.
- Programación de un temporizador que permita establecer el tiempo de muestreo.
- Otros cambios que permitan mantener el microcontrolador en funcionamiento aunque el terminal móvil no esté conectado.

Finalmente, en la librería se llevarán a cabo los siguientes ajustes:

- Reflejar los cambios en el protocolo que se lleven a cabo en el firmware.
- Creación de nuevos métodos en la interfaz que permitan el uso de las nuevas funcionalidades en la aplicación.

1.3 Estructura del proyecto

En este apartado se detallarán los contenidos que cubren la memoria, realizando una breve descripción de lo que abarca cada capítulo.

1. Introducción. En este capítulo se abordan las motivaciones por las que se ha elegido este proyecto y los objetivos que se desean cumplir con su elaboración.
2. Desarrollos existentes. Aquí se tratarán las alternativas existentes al hardware y software que se ha utilizado en el proyecto y se detallará por qué finalmente se ha empleado esta elección. Asimismo, se examinarán las herramientas utilizadas para el desarrollo.
3. Desarrollo. Este es el capítulo central del proyecto, donde se detallará el esquema propuesto para crear el sistema de monitorización de señales y se comentarán las modificaciones software llevadas a cabo para cumplir con los requisitos del proyecto. Asimismo, se describirá la aplicación Android que se ha creado.
4. Pruebas y validación. Este capítulo da cuenta de los distintos tipos de prueba llevadas a cabo para comprobar el funcionamiento adecuado del software y su correcta integración con el hardware.
5. Conclusiones y desarrollos futuros. En este apartado se recogerá la problemática encontrada a lo largo del desarrollo del proyecto, así como las posibles mejoras que, en un futuro, se podrían diseñar y aplicar al proyecto.

2 DESARROLLOS EXISTENTES

Si ya sabes lo que tienes que hacer y no lo haces
entonces estás peor que antes.

- Confucio -

Los teléfonos inteligentes pueden ser una de las mejores plataformas para el desarrollo de aplicaciones, debido al alcance que estos tienen en la población (8 de cada 10 teléfonos son smartphones [1]). Asimismo, en la mayoría de los casos, programar una aplicación móvil no es una tarea realmente difícil si se tiene en cuenta que la mayoría de Sistemas Operativos proporcionan su propio IDE y además es posible consultar la API de cualquiera de ellos, pudiendo acceder de esta forma a las diferentes funcionalidades del terminal.

Los dispositivos móviles constan hoy en día de multitud de sensores que han abierto un abanico de nuevas aplicaciones. A partir de estos es posible medir movimiento, orientación o condiciones ambientales. Entre los sensores más importantes destacan: acelerómetro, orientación, luz, presión, campo magnético, giroscopio, proximidad y temperatura. En función de cómo obtengan los datos, existen dos tipos de sensores:

- Sensores basados en hardware: son componentes físicos integrados en el terminal que obtienen los datos midiendo directamente propiedades ambientales. Por ejemplo: sensores de luz, proximidad, presión, temperatura interna, etc.
- Sensores basados en software: no disponen de elementos físicos aunque imitan a sensores basados en hardware. Obtienen sus datos de uno o más sensores basados en hardware. Por ejemplo: sensor de aceleración lineal, gravedad, orientación, etc.

Por otro lado, en función de su cometido existen los siguientes tipos de sensores:

- Sensores de movimiento: miden fuerzas de aceleración y fuerzas rotacionales alrededor de tres ejes (acelerómetros, sensores de gravedad, giróscopos, etc.).
- Sensores ambientales: miden varios parámetros ambientales como temperatura, presión, iluminación y humedad (barómetros, fotómetros y termómetros).
- Sensores de posición: miden la posición física del dispositivo (sensores de orientación y magnetómetros).

Sin embargo, a pesar de contar con todos estos sensores, en muchos casos no es suficiente para poder realizar funciones más avanzadas. Para solucionar esto se va a proponer en este trabajo el uso de IOIO, que proporcionará la interfaz necesaria para poder programar aplicaciones móviles que hagan uso del hardware externo de un microcontrolador.

En este capítulo se describirán los distintos Sistemas Operativos móviles más importantes del mercado, centrándonos especialmente en Android, el SO en el que se centrará este trabajo. Además, se va a detallar el funcionamiento de IOIO, desde su FW hasta la API de Android. Se buscarán también otros sistemas similares que puedan proporcionar algo parecido a IOIO. Finalmente, se explicarán las herramientas necesarias para programar una aplicación en IOIO.

2.1 Android y otros Sistemas Operativos móviles

2.1.1 Android

Android [2] es un SO móvil basado en Linux y diseñado para ser usado en dispositivos táctiles tales como teléfonos inteligentes o tablets, además de relojes inteligentes o televisores. Inicialmente, fue desarrollado por Android Corporation que fue comprada por Google en 2005. Fue presentado en 2007, y desde que en 2008 saliera a la venta el primer terminal con este SO, no ha parado de crecer, siendo los terminales que incorporan Android los que más se venden en todo el mundo [3].

Entre las características y servicios que puede ofrecer Android destacan [4]:

- Diseño de dispositivo: la plataforma es adaptable a pantallas de diferente resolución.
- Almacenamiento: uso de SQLite para almacenamiento de datos.
- Conectividad: soporte de diferentes tecnologías como GSM, UMTS, LTE, Wi-Fi, Bluetooth, entre otras.
- Soporte de Java: la mayoría de aplicaciones están escritas en Java. A pesar de ello, no hay una Máquina Virtual Java en Android, si no que primero se compila en un ejecutable Dalvik y corre en la Máquina Virtual Dalvik, diseñada específicamente para Android. De esta forma, se optimiza el uso de batería, memoria y procesador.
- Soporte para hardware adicional: Android soporta multitud de sensores (acelerómetros, giroscopios, magnetómetros, sensores de luz, presión, temperatura, etc.) además de hardware adicional mediante la interfaz USB.
- Multi-táctil: soporte para pantallas capacitivas con capacidad multi-táctil.
- Multitarea: multitarea real de aplicaciones.
- Reutilización de componentes: cualquier aplicación puede publicar sus capacidades y usar las capacidades de otra.
- Interfaz gráfica: la interfaz de usuario de Android está basada en XML. Mediante el uso de layouts (ficheros XML) se controla la posición de los elementos en pantalla.
- Entorno de desarrollo integrado: Android Studio es el IDE oficial para el desarrollo de aplicaciones Android. Incluye, entre otras cosas, un emulador de dispositivos, herramientas para la depuración de memoria y análisis del rendimiento del software.

La arquitectura del sistema es la siguiente:

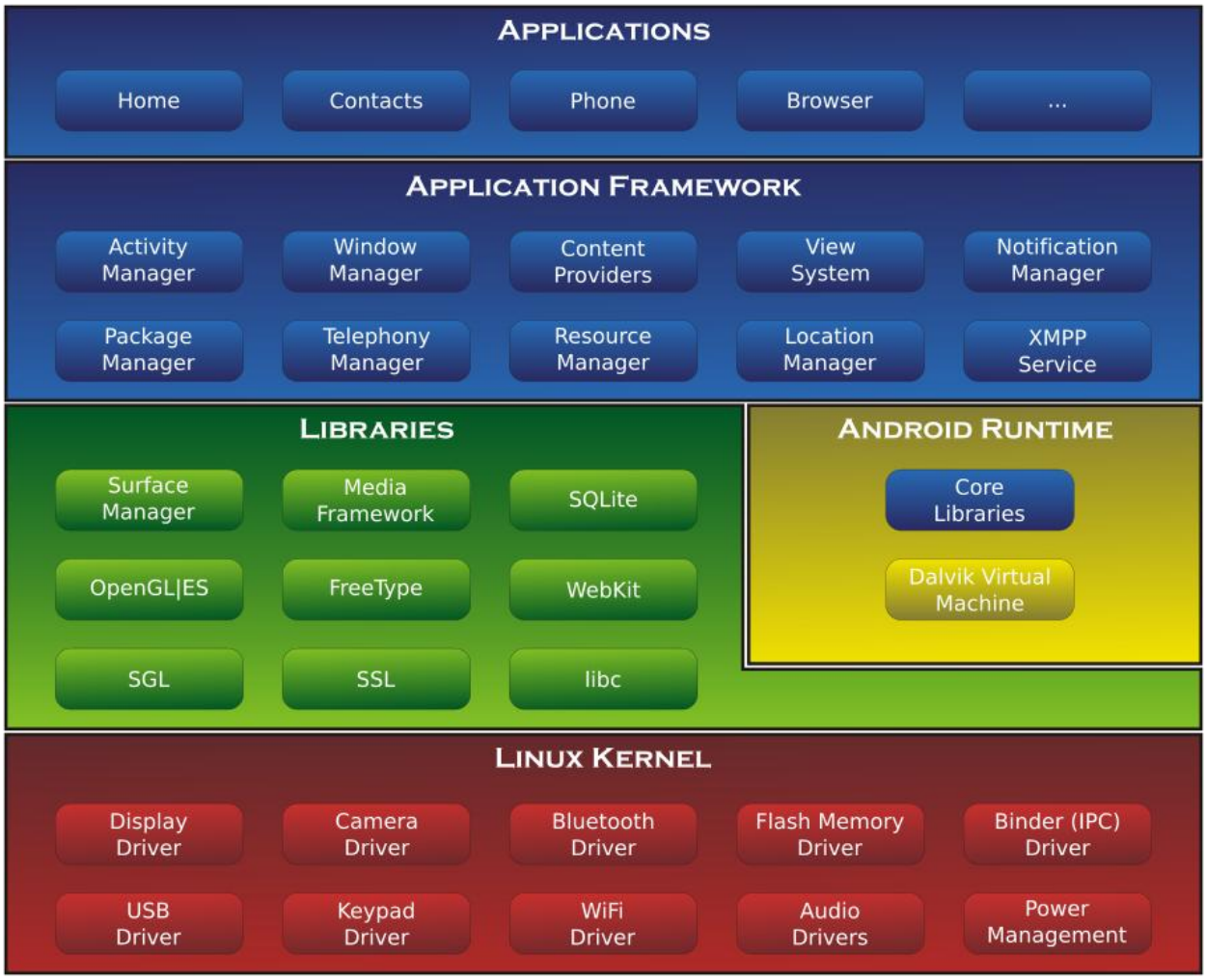


Figura 2-1. Arquitectura del sistema Android. Fuente: [2]

- *Linux Kernel* (Núcleo de Linux): el kernel o núcleo del SO (capa de abstracción entre el hardware y el software) de Android está basado en Linux. De esta forma, Android aprovecha servicios base del sistema como seguridad, gestión de memoria, gestión de procesos o la pila de red y el modelo de drivers.
- *Libraries* (Librerías Android): Android incluye un conjunto de librerías escritas en C/C++ usadas por varios componentes del sistema (Application Framework). Entre los componentes que incluyen estas librerías podemos destacar: Surface Manager (gestión de acceso a la pantalla), Media Framework (reproducción de imágenes, audio y vídeo) o SQLite (pequeña base de datos relacional), entre otros.
- *Android Runtime*: Android posee un conjunto de librerías base (Core Libraries) que proporcionan la mayor parte de las funciones disponibles en las bibliotecas base del lenguaje Java. Cada aplicación Android utiliza el kernel de Linux para su ejecución, con su propia instancia de la máquina virtual Dalvik. Esta tiene el código preparado teniendo en cuenta la duración de la batería y la limitación de memoria del terminal. De esta forma, Dalvik ejecuta aplicaciones en formato .dex (Dalvik Executable), el cual está optimizado para utilizar la mínima memoria.
- *Application Framework* (Framework de aplicaciones): proporciona una plataforma abierta para el desarrollo que permite la reutilización de componentes. De esta forma, a través del framework, el desarrollador puede acceder a los dispositivos, información de ubicación, ejecutar servicios, etc.

- *Applications* (Capa de aplicaciones): en esta capa se ubicarán las aplicaciones preinstaladas y creadas por el desarrollador. Todas estas aplicaciones están escritas en lenguaje Java. Por defecto se incluyen aplicaciones como el cliente email, gestor SMS, navegador, contactos y Play Store.

Las aplicaciones Android tienen componentes que el sistema puede instanciar y ejecutar por separado cuando sea necesario. Existen cuatro tipos de componentes:

- Actividades: componente básico de una aplicación Android. Cada actividad representa una tarea que la aplicación puede realizar, normalmente apoyada en la correspondiente pantalla como interfaz con el usuario. Este será el componente que se utilizará para la recepción de los datos de las señales de IOIO en Android.
- Servicios: un servicio es un componente que se ejecuta en segundo plano para realizar tareas durante un largo periodo de tiempo.
- Receptores de mensajes de difusión: componentes que responden ante los mensajes de difusión. Estos pueden ser generados por el sistema (por ejemplo: batería baja, cambio de zona horaria, …) o por otro componente (por ejemplo: forzar la actualización de ciertos valores en una interfaz gráfica, informe de descarga de datos, …).
- Proveedores de contenido: permiten que un conjunto de datos estén a disposición de otras aplicaciones. Estos datos pueden estar en el sistema de ficheros, en una base de datos SQLite o en otros medios.

Los componentes de la aplicación se declaran en el archivo "manifiesto" (`AndroidManifest.xml`). En este fichero, además, se declaran entre otras cosas: el icono de la aplicación, el mínimo SDK y los permisos. Aquí se puede ver un ejemplo:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.dinel.javi.prueba2activities" >

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".PruebaIOIOActivity"
            android:label="@string/title_activity_prueba_ioio" >
        </activity>
    </application>

</manifest>
```

Para conocer mejor la estructura de una aplicación Android, es necesario conocer la organización de los archivos de la que está formada. La estructura es la siguiente:

- Fichero `AndroidManifest.xml`: contiene la definición de la aplicación.
- Directorio `src/`: contiene los ficheros fuente (.java) que contienen la lógica de la aplicación.
- Directorio `res/`: contiene un grupo de directorios que almacena los recursos necesarios para la aplicación (imágenes, interfaces gráficas, etc.). El más importante de ellos es el directorio `layout/`, donde se almacenan los ficheros que definen las interfaces gráficas del usuario.

Para programar una aplicación en Android hay que programar una Actividad, que es el componente básico. A cada actividad se le asigna una ventana por defecto (puede llenar toda la pantalla o estar sobre otras) y se puede iniciar de las siguientes formas:

- Al inicio de la aplicación, indicándolo en el archivo manifiesto.
- Mediante otra Actividad, pasando a ser la Actividad de la aplicación.

El ciclo de vida de una Actividad es el siguiente:

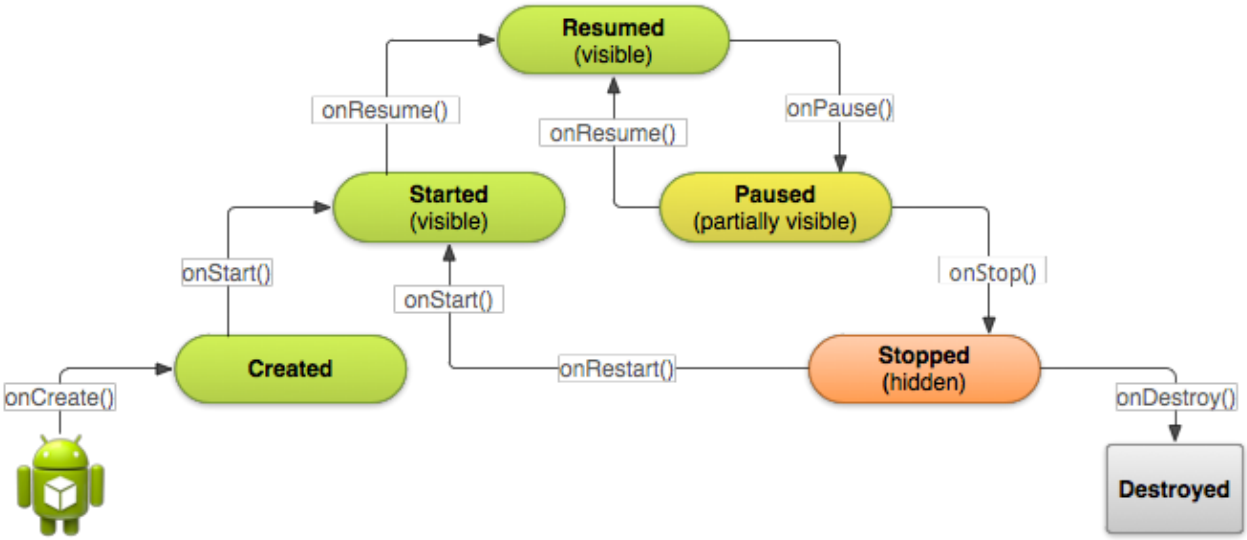


Figura 2-2. Ciclo de vida de una actividad Android. Fuente: [2]

- Ciclo de vida: entre los métodos callback `onCreate()` y `onDestroy()`. El primero de ellos da el contenido inicial a la actividad y solicita los recursos que necesite, mientras que el segundo libera los recursos ocupados por la actividad.
- Entre las funciones `onStart()` y `onStop()`: tiempo de vida en el que el usuario puede interactuar con la actividad.
- Entre las funciones `onPause()` y `onResume()`: tiempo de vida en el que la actividad no está completamente disponible para el usuario, cediendo el control parcialmente a otra actividad.

Para iniciar la ejecución de otra actividad o componente, Android proporciona los intent, un elemento que representa el intento de un componente de iniciar la ejecución de otro. Es una forma de pasar mensajes. Ejemplo para iniciar una actividad de nombre `IOIOActivity`:

```
Intent intent = new Intent(this, IOIOActivity.class);
```

Asimismo, mediante objetos de la clase `Intent` también podemos enviar información adicional al nuevo componente que se va a ejecutar.

Cuando se invoca el primer componente de una aplicación Android se inicia un proceso con un único hilo de ejecución. Por defecto, todos los componentes de la aplicación se ejecutan en ese hilo, denominado main. Este hilo se encarga principalmente de la interfaz gráfica de la aplicación, incluyendo el tratamiento de cada uno de los eventos que se provocan en la interfaz de usuario, como por ejemplo presionar un botón.

Si el hilo principal se bloquea, se obtiene el mensaje ANR (*Application Not Responding*), por lo que no se puede bloquear mucho más de unos pocos segundos. De esta forma, para realizar una tarea que podría bloquear el hilo principal, es necesario utilizar un hilo independiente. Para ello se usan las clases `Thread` y `Runnable`.

2.1.2 iOS

iOS es el SO móvil de Apple que utilizan los dispositivos iPhone, iPad y iPod Touch. Es el segundo SO móvil en cuanto a cuota de mercado [5].

Este SO deriva de Mac OS X, que a su vez está basado en Darwin BSD, un sistema operativo de tipo Unix. iOS está estructurado en capas. Las inferiores tienen servicios y tecnologías fundamentales y las superiores se apoyan en las inferiores para ofrecer servicios más sofisticados. A la hora de programar, se puede utilizar cada capa, aunque las capas superiores suelen ofreces abstracciones de las capas inferiores, simplificando su uso.

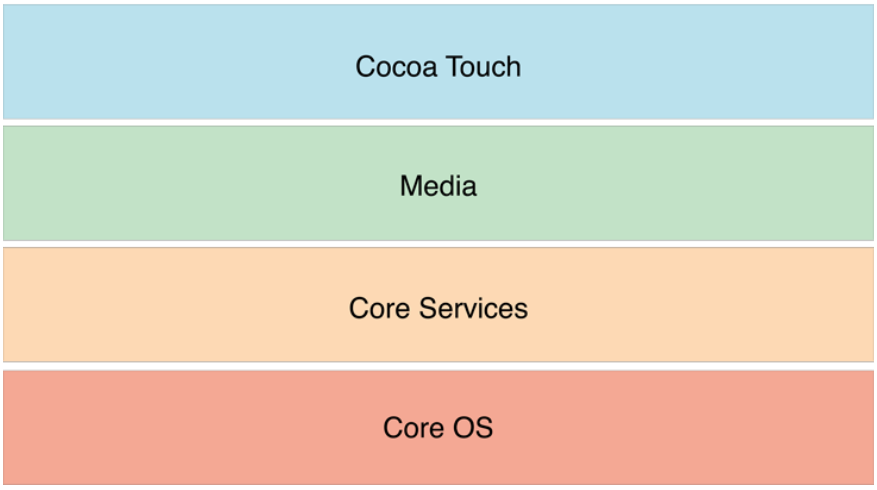


Figura 2-3. Estructura en capas de iOS. Fuente: [2]

Objective-C es el lenguaje de programación principal de iOS. Es un lenguaje orientado a objetos creado como un superconjunto de C. Tiene algunas extensiones respecto a C como clases y mensajes o algunos tipos nuevos. Asimismo, permite la definición de protocolos que definen comportamientos comunes para clases distintas.

2.1.3 Windows 10 Mobile

Windows 10 Mobile es el SO móvil de Microsoft para teléfonos inteligentes. Tras la desaparición de Windows Mobile y Windows Phone, esta es la apuesta de Microsoft para mejorar su posición en el mercado.

Posee una plataforma (UWP) [6] que permite crear aplicaciones universales para Windows (PC, tablet o móvil). Asimismo, se pueden utilizar varios lenguajes de programación como JavaScript, C#, Visual Basic o C++.

2.1.4 Blackberry OS

Blackberry OS es uno de los sistemas operativos más populares para corporaciones, ya que está claramente orientado a su uso profesional como el gestor de correo electrónico y la agenda.

Permite el desarrollo de aplicaciones de terceros, pero para acceder a ciertas funcionalidades se necesita ser firmado digitalmente.

2.2 Interfaz externa: IOIO

IOIO [7] [8] es una placa electrónica que consta de un microcontrolador PIC, una interfaz USB y otros componentes electrónicos, originalmente diseñada para programar aplicaciones en Android. Existen dos tipos: IOIO y IOIO-OTG, siendo la única diferencia entre ambas que IOIO-OTG puede trabajar tanto con dispositivos Android como con PCs. En ambos casos, la conexión con el terminal se puede realizar tanto por USB como por Bluetooth, pudiendo interactuar de esta forma a partir de una aplicación Android mediante una API de Java que permite programar las funciones del microconrolador con un lenguaje de alto nivel.

El actual Firmware de IOIO ofrece, entre otras funcionalidades: entrada/salida digital, conversión A/D, salida PWM, comunicación I^2C, SPI o UART, entre otras. De esta forma, mediante un dispositivo Android es posible aprovechar esta serie de funcionalidades programando un microcontrolador mediante un lenguaje de alto nivel, gracias a la librería que se proporciona.

Figura 2-4. Logotipo de IOIO. Fuente: [7]

IOIO-OTG consta de los siguientes elementos:

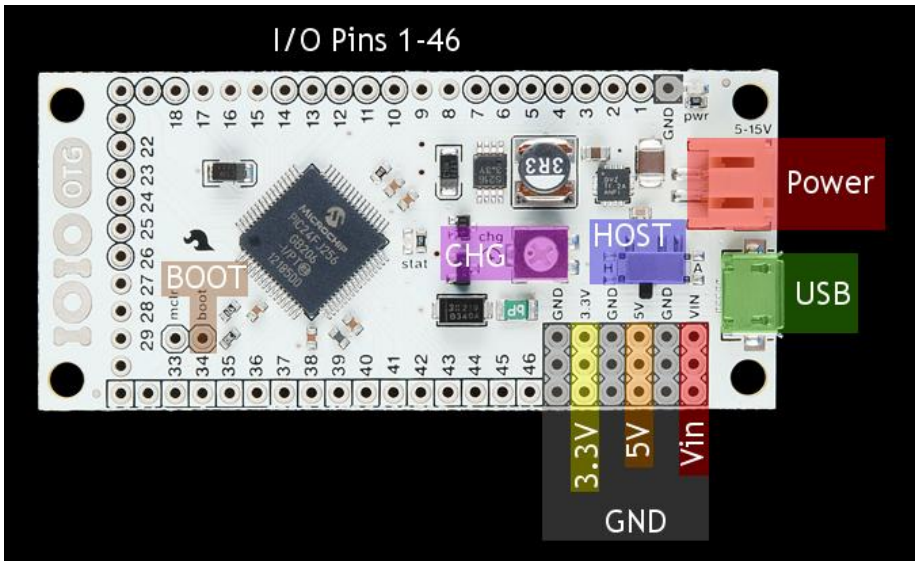


Figura 2-5. IOIO-OTG. Fuente: [7]

- Conector USB (micro-AB hembra): mediante este es posible conectar IOIO al terminal Android o PC, o bien un dongle Bluetooth para realizar la comunicación sin cables.
- Conector de alimentación (2 pines JST hembra): usado para la alimentación de IOIO. Se debe suministrar una tensión de entre 5 y 15 V y 1 A. En nuestro caso se ha adaptado un conector USB a un JST para de esta forma poder alimentar IOIO conectándolo simplemente a un ordenador, por ejemplo.
- Pines a tierra (GND): IOIO dispone de diez pines para las conexiones a tierra.
- Pines Vin: estos tres pines se usan para la alimentación de la placa, como alternativa al conector JST.
- Salidas a 5 V: tres pines con salida a 5 V.
- Salidas a 3,3 V: tres pines con salida a 3,3 V.
- Pines de entrada/salida: IOIO dispone de 46 pines que pueden ser usados para varias finalidades (entrada/salida digital, conversión A/D, comunicación I^2C, etc.).
- Led PWR (rojo): cuando está encendido indica que IOIO está conectada a la corriente.
- Led STAT (amarillo): led de propósito general para uso en aplicaciones.
- Pin MCLR: para programar en IOIO un nuevo firmware de inicio (*bootloader*).
- Pin BOOT: se utiliza para poner en modo de inicio (*bootloader*) IOIO. A partir de este, como se explicará más adelante, es posible modificar el firmware del microcontrolador.
- *Charge Current Trimmer* (CHG): ajusta la cantidad de corriente suministrada cuando IOIO actúa como USB host.
- *Host switch*: cuando la pestaña está en "A", IOIO detecta automáticamente el modo en el que debe actuar en función de si se conecta un USB micro-A o micro-B. Si la pestaña está en "H" fuerza a IOIO a actuar en modo Host.

2.2.1 Comunicación IOIO-Android

La comunicación entre el dispositivo Android y el microcontrolador puede realizarse bien por USB, utilizando el microcontrolador como USB host y el terminal como USB slave (debe poder soportarlo), o mediante Bluetooth. Para la comunicación entre el teléfono y el microcontrolador, IOIO implementa un protocolo de comunicación mediante el cual se establece una conexión mediante sockets TCP con el ADB a través del puerto 4545. Así, mediante un paso de mensajes (estos ya están definidos en el firmware del microcontrolador) entre la aplicación y el microcontrolador, es posible programar a este con la librería que se proporciona.

Es por esto último por lo que en el AndroidManifest es necesario añadir los permisos para Internet y Bluetooth (para Bluetooth porque la librería de IOIO intenta establecer la conexión mediante este protocolo). De esta forma, en cualquier programa de Android, hay que añadir las siguientes líneas al AndroidManifest:

AndroidManifest.xml

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

Una vez que se pierde la conexión con el terminal el protocolo y el estado de los periféricos se resetea inmediatamente.

2.2.2 Firmware de IOIO

Actualmente, el firmware de IOIO está formado por dos programas separados, ambos programados en la memoria flash interna, pero en diferentes espacios de memoria.

El primero de ellos es el firmware de aplicación, que es el que proporciona el control de IOIO y sus periféricos y permite la comunicación con el dispositivo Android. Este programa está en constante desarrollo y es el que se modificará en este trabajo, añadiendo nuevas funcionalidades.

El segundo es el *bootloader* o gestor de arranque. Su funcionalidad es básica y sencilla: permitir la actualización del firmware de aplicación. Cada vez que se inicia IOIO, lo primero que se ejecuta es el *bootloader*. Este comprueba si el pin BOOT de IOIO está conectado a tierra (GND). Si no está conectado, ejecuta el firmware de aplicación y desaparece inmediatamente. En caso contrario, el *bootloader* detectaría que el pin BOOT está conectado a tierra y IOIO entraría en modo *bootloader*. Así, es posible instalar un nuevo firmware de aplicación en IOIO.

El firmware de aplicación de IOIO está organizado en una serie de ficheros escritos en lenguaje C, en los que se programa el microcontrolador, de forma que actúe ante una serie de mensajes que tienen lugar en la comunicación entre el terminal y IOIO. La estructura del código y los principales módulos del firmware son los siguientes:

- Fichero main.c: contiene la máquina de estados del microcontrolador y trata de establecer la conexión con el terminal Android.
- Módulo protocol.{h, c}: se encarga de analizar los mensajes del protocolo entrantes, remitiéndolos al módulo encargado de manejar la función que dicte el mensaje. También se encarga de los mensajes de salida, proporcionando el almacenamiento de los datos sin que los demás módulos tengan que preocuparse sobre si el canal de salida está actualmente ocupado.
- Módulo features.{h, c}: contiene funciones generales (reseteo de IOIO, modo de los pines, etc.).
- Módulo adc.{h, c}: implementa las funciones encargadas de la conversión analógica/digital.
- Módulo pwm.{h, c}: implementa las funciones encargadas de generar una salida PWM.
- Módulo digital.{h, c}: implementa la entrada/salida digital.
- Módulo pins.{h, c}: contiene la información sobre el mapeado del número de los pines, ya que el número de los pines sobre la placa no es el mismo que los registrados en el microcontrolador.

2.2.2.1 Actualización del Firmware en IOIO

Para actualizar el Firmware de IOIO es necesario que esté en modo bootloader. Para conseguir esto hay que seguir los siguientes pasos:

1) Con IOIO apagado, se conecta el pin BOOT a tierra (GND).
2) Utilizando un USB micro-B conectamos IOIO al ordenador en el que se vaya a actualizar el firmware. Entonces, IOIO debería encenderse (led PWR encendido) y además se encenderá el led amarillo (STAT).
3) Por último, hay que quitar la conexión del pin BOOT a tierra. Tras esto, el led STAT debe parpadear indicando que IOIO está en modo bootloader.

Una vez hecho esto es necesario un programa que escriba el nuevo firmware en la memoria flash del microcontrolador. Para ello, se dispone de IOIODude, que se puede obtener a través del siguiente enlace: https://github.com/ytai/ioio/raw/master/release/apps/IOIODude-0102.zip.

Tras descomprimir el archivo, es necesario tomar el script ioiodude y darle permisos de ejecución. En Linux:

```
#:~$ chmod a+x ioiodude
```

De esta forma, si se ejecuta el script ahora se podrán observar las opciones que ofrece IOIODude:

```
IOIODude V1.0

Usage:
ioiodude <options> versions
ioiodude <options> fingerprint
ioiodude <options> write <ioioapp>

Valid options are:
--port=<name> The serial port where the IOIO is connected.
--reset Reset the IOIO out of bootloader mode when done.
--force Bypass fingerprint matching and force writing.
```

Una vez que IOIO está en modo bootloader, por seguridad, se comprueba que efectivamente se encuentra en este estado ejecutando el siguiente comando:

```
#:~$ ./ioiodude --port=/dev/tty.usbmodem1411 versions;
```

siendo /dev/tty.usbmodem1411 el puerto USB al que está conectado IOIO. Entonces, la respuesta debería ser la siguiente:

```
IOIO Bootloader detected.

Hardware version: SPRK0020
Bootloader version: IOIO0400
Platform version: IOIO0030
```

Si no estuviera en modo bootloader, la respuesta del comando sería esta:

```
IOIO Application detected.

Hardware version: SPRK0020
Bootloader version: IOIO0400
Platform version: IOIO0030
```

Finalmente, para cargar el nuevo firmware en la memoria flash del microcontrolador solo habrá que ejecutar el siguiente comando:

```
#:~$ ./ioiodude --port=/dev/tty.usbmodem1411 --reset write firmware.ioioapp;
```

Y se tendrá la siguiente respuesta:

```
Comparing fingerprints...
Fingerprint mismatch.
Writing image...
[########################################]
Writing fingerprint...
Done.
```

2.2.3 Librería Android

IOIOLib es una colección de librerías, tanto para Android como para PC, que permite la programación del microcontrolador de IOIO y sus periféricos a partir de una aplicación. De esta forma, a partir de las interfaces Java podemos cubrir numerosas características de la placa.

Las dos principales librerías de IOIOLib son `IOIOLibPC` y `IOIOLibAndroid`, usadas para utilizar IOIO como interfaz en PC o Android, respectivamente. Asimismo, `IOIOLibBT` y `IOIOLibAccessory` complementan a `IOIOLibAndroid` proporcionando las funcionalidades de comunicación mediante Bluetooth o *Android Open Accessory*, respectivamente. Para este proyecto se desarrollará una aplicación

Android, por lo que nos centraremos en la librería dedicada a este SO.

Las librerías se organizan en varios paquetes Java:

- Paquete `ioio.lib.api`: contiene la API (IOIOLib Core API) para controlar IOIO. Este paquete será el que use la aplicación.
- Paquete `ioio.lib.impl`: contiene la implementación de las interfaces.
- Paquete `ioio.lib.util`: contiene el Framework de aplicación de IOIOLib (IOIOLib Application Framework), que permitirá programar aplicaciones para IOIO de forma más fácil.

2.2.3.1 IOIOLib Core API

La API de IOIO es la que permite la interacción de la aplicación con IOIO. Contiene una representación de la placa (vía la interfaz `IOIO`), así como otra serie de interfaces que proporcionan acceso a algunas características del microcontrolador y sus periféricos.

La interfaz `IOIO` es el corazón de la librería. Una instancia de esta interfaz representa una placa IOIO física, que mediante los métodos que posee permite acceder a las funciones del microcontrolador. Obtener una instancia de la interfaz `IOIO` depende, entre otros factores, del tipo de aplicación (Android o PC) o del tipo de comunicación (USB o Bluetooth) que se esté llevando a cabo. Para ocultar esta complejidad se proporciona el Framework de aplicación de IOIOLib.

De esta forma, una vez que IOIO está conectada y se tiene una instancia de la interfaz `IOIO` se podrá acceder a las diferentes funciones y recursos que la placa proporciona: entrada/salida digital, conversión analógica/digital, salida PWM, etc. Cuando IOIO se desconecta o se sale de la aplicación, se lanza la excepción `ConnectionLostException`, entonces la instancia de IOIO muere. Así, si se restablece la conexión, se creará una nueva instancia.

Por otro lado, para conseguir el reinicio de la placa sin tener que desconectar la alimentación, es decir, mediante software, existen dos tipos de reset: suave (*soft*) y fuerte (*hard*).

Con un *hard reset* se consigue exactamente lo mismo que quitando la alimentación y volviéndola a poner. Así, la instancia de `IOIO` se elimina y tendrá lugar un completo reinicio, pasando incluso por el *bootloader*.

Mediante el *soft reset* todo vuelve al estado inicial y se liberan todos los recursos. De esta forma, la conexión con IOIO no se pierden, pero sí se desactivan todos los módulos y se cierran todas las instancias obtenidas a partir de la instancia `IOIO`.

2.2.3.2 IOIOLib Application Framework

Cuando se inicia una aplicación de IOIO, el Framework determina todas las posibles conexiones de la placa y por cada una de ellas, proporciona el método `createLooper()`, mediante el cual el cliente podrá interactuar con IOIO.

`IOIOLooper` es una interfaz que permite notificar al cliente cuándo se ha establecido o cerrado una conexión con IOIO. Por cada IOIOLooper devuelto por el cliente, el Framework creará un hilo para la ejecución de los procesos relativos a IOIO. De esta forma, se evita bloquear el hilo principal de la aplicación Android. Aproximadamente, este hilo sigue el siguiente razonamiento:

1) Espera a que se establezca una conexión con IOIO.
2) Comprueba que la versión del FW de IOIO es compatible con la librería Android.
3) Si es compatible:
   - Llama al método `setup()` de `IOIOLooper`. En este método normalmente se realiza la apertura de pines o se inicializa alguna conexión.
   - Llama repetitivamente al método `loop()` de `IOIOLooper`. Aquí se realizará la tarea que se desee con IOIO.
4) Si no es compatible:

- Llama al método `incompatible()` de `IOIOLooper`. En este método se puede lanzar un aviso al cliente en la aplicación sobre la incompatibilidad del FW y la librería.

5) Una vez que se suprime la conexión se llama al método `disconnect()` y la instancia de `IOIO` deja de estar disponible.

En resumen, la estructura de una aplicación Android para IOIO sería la siguiente:

```
public class EjemploIOIOActivity extends IOIOActivity {

  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_medir_temperatura);
  }

  class Looper extends BaseIOIOLooper {
      protected void setup() {

      }

      public void loop() {

      }
  }

  protected IOIOLooper createIOIOLooper() {
      return new Looper();
  }
}
```

Como se puede observar, es necesario que la actividad sea de tipo `IOIOActivity` para implementar todos los métodos que permitan la programación de IOIO.

2.2.4 Inconvenientes del actual Firmware/Librería

El Firmware de IOIO brinda una gran cantidad de posibilidades a la hora de programar un microcontrolador a partir de una aplicación móvil, pero también está limitado en otros aspectos que hacen que no se tengan las mismas oportunidades que programando un microcontrolador directamente.

Uno de esas características es la posibilidad de que el microcontrolador funcione de forma independiente al terminal cuando sea necesario. Por ejemplo, para monitorizar una señal analógica durante un largo periodo de tiempo. Con el Firmware actual, para poder realizar esto, el terminal Android debe estar conectado permanentemente a IOIO, lo que hace que sea algo tedioso si la señal necesita estar todo el tiempo monitorizada.

Se podría pensar que para alguna tarea como esta podría ser útil programar IOIO desde el *smartphone*, desconectarlo y, tras el tiempo que se considere necesario, volver a conectar el terminal y recuperar los datos. Para ello, sería necesario modificar el firmware, añadiendo nuevos mensajes y funciones, así como la librería Android. De esta forma, se resolvería uno de los mayores inconvenientes de IOIO.

2.3 Sistemas similares

Además de IOIO y IOIO-OTG existen otras placas y sistemas que pueden funcionar de forma similar a IOIO, integrándose también con Android. En este apartado repasaremos algunos de los más importantes.

2.3.1 Seeduino ADK Main Board

La placa *Seeduino ADK Main Board* [9] realiza la comunicación con el terminal Android mediante USB aprovechando el protocolo *Open Accessory* de Android. Es compatible con dispositivos Android v1.5 usando *MicroBridge* y con los v2.3.4 o superiores mediante el uso de la API de Google *Open Accessory* (ADK).

Figura 2-6. Seeduino ADK Main Board. Fuente: [9]

Entre las características de esta placa destacan las siguientes:

- 56 entradas/salidas digitales
- 16 entradas analógicas
- 14 salidas PWM
- Conector Jack para alimentación entre 6 y 18 V
- Conector USB

2.3.2 Andruino (Android + Arduino)

Arduino [10] es una plataforma de código abierto basada en una placa con microcontrolador y con un entorno de desarrollo propio para la creación de software con el microcontrolador. Consta de un puerto USB conectado a un módulo adaptador USB-Serie que permite programar el microcontrolador desde cualquier PC y realizar pruebas de comunicación.

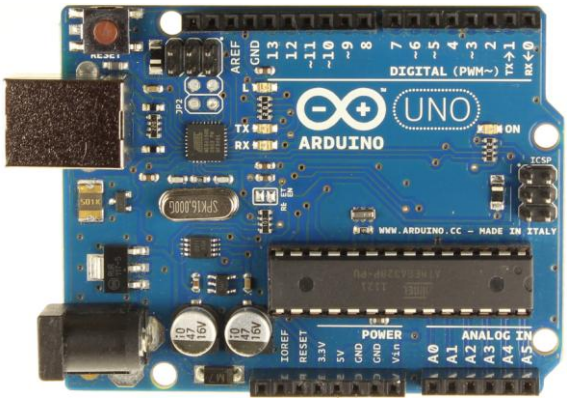


Figura 2-7. Arduino UNO. Fuente: [10]

Algunas de sus características son las siguientes:

- 14 entradas/salidas digitales
- 6 salidas PWM
- 6 entradas analógicas
- 2 pines (SDA y SCL) para comunicación I^2C
- Conector Jack para alimentación entre 7 y 12 V

Arduino ofrece además una serie de bibliotecas estándar escritas en C/C++ que proporcionan funcionas para utilizar la memoria EEPROM, conectar a internet mediante Ethernet Shield y controlar motores, entre otras.

Aunque en principio no existe una librería que proporcione funciones para la comunicación con un terminal Android, es posible realizar esta comunicación mediante una conexión a Internet o a la red local utilizando el Ethernet Shield de Arduino [11]. A partir de aquí, solo se necesita programar la aplicación Android mediante Appcelerator [12] para la programación multiplataforma mediante Javascript.

Concretamente, creando un cliente HTTP que se comunique con Arduino se conseguirá la comunicación entre Arduino y el dispositivo móvil.

2.3.3 PhoneDrone Board

PhoneDrone [13] es otra placa electrónica que nos permite conectar cualquier terminal Android con versión igual o superior a la 2.3.4, especialmente diseñado para su uso en drones y vehículos no tripulados.

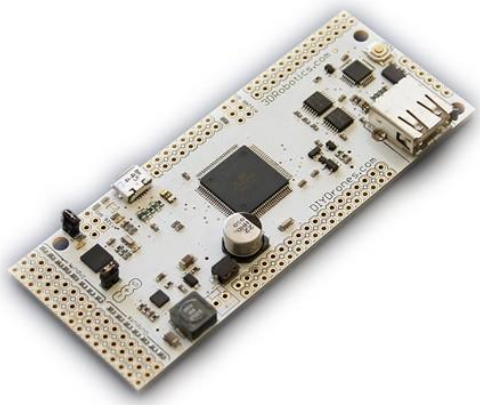


Figura 2-8. PhoneDrone Board. Fuente: [13]

Sus principales características son:

- Gran variedad de entradas/salidas digitales, entradas analógicas, salidas PWM, etc.
- Tensión de entrada de entre 6 y 36 V
- Conector USB
- Compatibilidad con Arduino

2.3.4 Microchip PIC24F Accessory Development Starter Kit

El kit de Microchip PIC24F [14] para el desarrollo de accesorios Android es otra de las opciones disponibles para programar microcontroladores mediante aplicaciones Android. Esta plataforma nos proporciona una biblioteca para el acceso a los dispositivos Android mediante el framework de las versiones 2.3.4, 3.1 y superiores de Android.

Figura 2-9. Placa de desarrollo Microchip PIC24F. Fuente: [14]

Las principales características de esta placa son las siguientes:

- Microcontrolador PIC24F 16-bit con USB-OTG
- Dispone de botones, potenciómetros y leds para la interfaz con el usuario
- Compatible con Arduino

2.4 Herramientas usadas

Como se ha visto en apartados anteriores, para modificar el comportamiento de IOIO es necesario modificar el firmware del microcontrolador, la librería de Android y, por tanto, la aplicación que se desarrolle. Es por ello que se van a necesitar varios entornos de desarrollo para modificar cada una de las partes.

En concreto, se necesitan tres IDEs:

- MPLAB: es el entorno de desarrollo de Microchip, necesario para programar el Firmware del microcontrolador PIC de IOIO.
- Eclipse: con este programa se modificará la librería de Android. También se puede utilizar para la programación de las aplicaciones Android, instalando previamente el plugin ADT (*Android Developer Tools*) y el SDK.
- Android Studio: es la plataforma sustituta de Eclipse, necesaria para desarrollar aplicaciones para Android.

En este apartado se describirá cada una de estas herramientas y cómo usarlas para desarrollar IOIO.

2.4.1 MPLAB X

MPLAB X [15] es el IDE de Microchip para el desarrollo de aplicaciones para microcontroladores. Fue desarrollado a partir del IDE de Oracle, NetBeans. Soporta la edición, depuración y programación de los microcontroladores PIC de 8, 16 y 32 bits. Es compatible con Windows, Linux y OS X y se puede obtener de forma gratuita desde su sitio web.

Para la modificación del firmware del microcontrolador, una vez descargado este entorno de desarrollo, habrá que importar la última versión del firmware, que será sobre la que se realicen las modificaciones. Para ello, hay que dirigirse a Files>Open Project…:

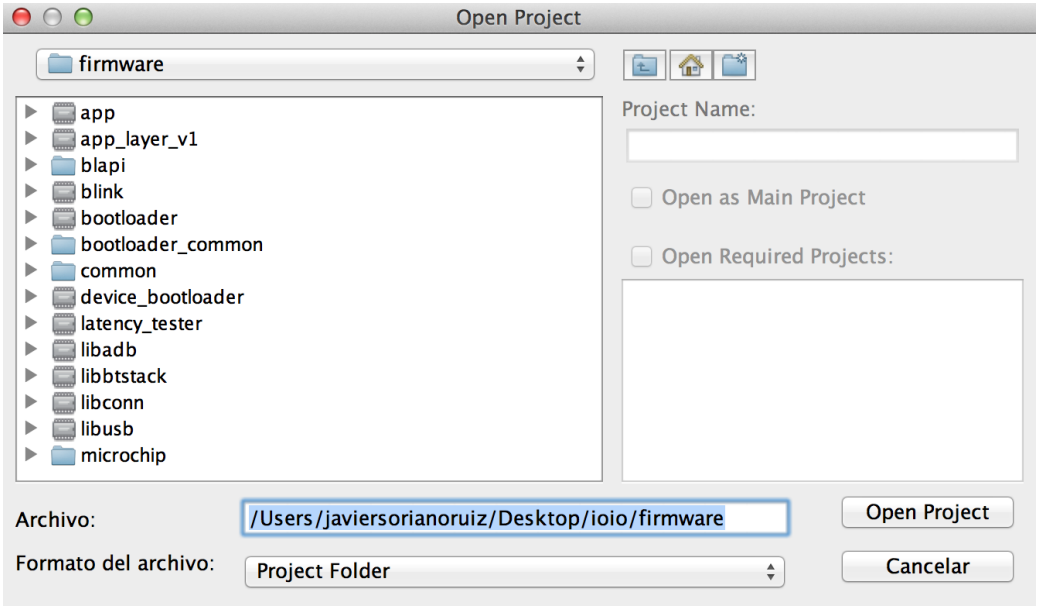


Figura 2-10. Importación del firmware a MPLAB X

El firmware de aplicación es el proyecto app_layer_v1, pero además de este hay que importar los proyectos libadb, libbtstack, libconn y libusb, necesarios para la compilación del firmware. También se incorporará el proyecto Blink, un firmware básico que se utilizará para realizar una serie de pruebas.

Una vez que los proyectos están incorporados a MPLAB X, es necesario configurar el compilador y otros aspectos para obtener el fichero de producción app_layer_v1_production.hex con el firmware de IOIO. Para ello, en cada uno de los proyectos hay que realizar lo siguiente:

1) Seleccionar el proyecto con el botón secundario y en Properties>Manage Configurations…, configurar IOIO0030 como "Set Active".
2) Dentro de la ventana principal de las propiedades hay que seleccionar como compilador el XC16.
3) En las opciones del compilador, en XC16 (Global Options), seleccionar xc16-gcc y en la pestaña de Option categories seleccionar Optimizations. Una vez ahí hay que establecer un Optimization level = 1.

Finalmente, tras hacer esto con cada uno de los proyectos que componen el firmware de IOIO se podrá compilar y generar el fichero de producción mediante la opción Run>Build Project (AppLayerV1). Este fichero se creará dentro del directorio app_layer_v1/dist/IOIO0030/production.

Ahora sólo faltaría convertir el fichero resultante .hex en un archivo .ioio, compatible con IOIO. Para ello, dentro del directorio del firmware en la carpeta tools, se nos proporciona la herramienta make-ioio-bundle. Así, ejecutando el siguiente comando se obtendrá el firmware ya preparado para actualizar en IOIO:

```
  #:~$ ./tools/make-ioio-bundle
/Users/javiersorianoruiz/Desktop/ioio/firmware/app_layer_v1/dist newFWIOIO
IOIO0030
```

2.4.2 Eclipse

Eclipse [16] es un IDE de código abierto que dispone de un editor de texto con resaltado de sintaxis, compilación en tempo real, control de versiones con CVS, asistentes para creación de proyectos, clases, test, etc., y refactorización. Lo que diferencia a Eclipse de otros IDEs es la contribución de sus usuarios y la posibilidad de añadir *plugins* (parches o módulos que añaden nuevas funcionalidades).

Para añadir la librería de IOIO para Android a Eclipse, simplemente hay que utilizar la opción File>Import…>Existing Projects into Workspace y seleccionar el directorio raíz del proyecto. Una vez hecho esto se incorporará el proyecto a Eclipse y no será necesario configurar nada más.

Esta librería habrá que incorporarla a la aplicación de Android, por tanto, cada vez que se realice un cambio se deberá crear un fichero de extensión .JAR que posteriormente habrá que incorporar al proyecto de Android Studio. Para realizar esto hay que seleccionar File>Export…>JAR file.

2.4.3 Android Studio

Android Studio [17] es el entorno de desarrollo integrado oficial para Android. Entre las características de este IDE destacan:

- Soporte para la construcción basada en Gradle.
- Herramientas Lint para detectar problemas de rendimiento, usabilidad, compatibilidad de versiones, y otros problemas.
- Plantillas para crear diseños comunes de Android.
- Consola de desarrollador con consejos de optimización, ayuda para la traducción y estadísticas de uso.

Como se comentó en el apartado anterior, para la programación de aplicaciones en Android es necesario incluir la librería IOIOLib generada con Eclipse con el formato .jar. Para incluirla hay que ir a la pestaña Project y arrastrar el fichero generado hasta la carpeta libs. Una vez ahí, es necesario seleccionar el fichero y marcar la opción "Add as a library", lo que permitirá que el Gradle reconozca la librería y se pueda hacer uso de las clases y métodos de IOIO.

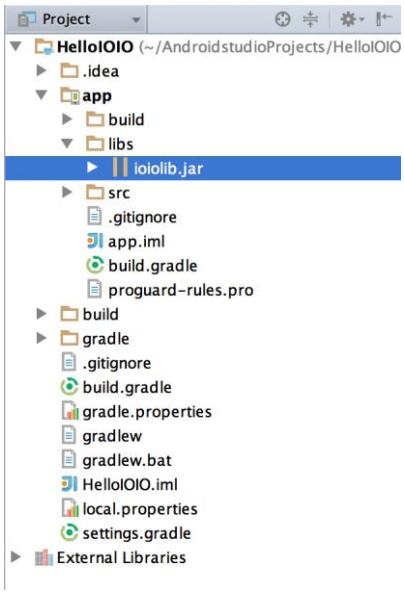


Figura 2-11. Librería IOIOLib en un proyecto de Android Studio

3 DESARROLLO

El cuerpo humano es el carruaje; el yo, el hombre que lo conduce; el pensamiento son las riendas, y los sentimientos los caballos.

- Platón -

Como se mencionó en el apartado 2.2.4 del capítulo anterior, el firmware que incorpora IOIO y la librería que se proporciona presentan como principal inconveniente la dependencia entre el móvil y el microcontrolador. Es decir, que, en principio, desde el terminal no es posible programar el microcontrolador para que realice una función independiente. Por ejemplo, si se desea monitorizar una señal, el terminal móvil deberá estar conectado y con la aplicación abierta durante toda la monitorización. Por tanto, en un principio, no será posible programar la monitorización, desconectar el móvil y, posteriormente, obtener los resultados.

Para implementar un sistema de monitorización de señales será necesario, por tanto, llevar a cabo una serie de modificaciones que permitan el muestreo de una señal analógica en el microcontrolador que funcione de forma independiente al terminal Android.

El objetivo es conseguir programar el muestreo de una determinada señal analógica desde el móvil, que el microcontrolador almacene los resultados en su memoria interna y que, tras un instante de tiempo determinado, se vuelva a conectar el terminal y el microcontrolador le devuelva los datos que ha registrado, observando en la aplicación la evolución de esa señal durante ese periodo de tiemp

3.1 Esquema propuesto

Para la comunicación con IOIO el terminal móvil y el microcontrolador deberán estar conectados durante el tiempo que dure el intercambio de mensajes mediante un cable microUSB. La monitorización de la señal se llevará a cabo por el pin 36 de IOIO, que recogerá la señal analógica y mediante el convertidor analógico digital la convertirá en una señal digital que almacenará en su memoria RAM. Así pues, mientras IOIO esté muestreando y no sea necesaria la comunicación con el terminal, estos no tendrán que estar conectados. Posteriormente, una vez que se desee obtener y visualizar los datos en el terminal, habrá un intercambio de mensajes entre el dispositivo Android y IOIO.

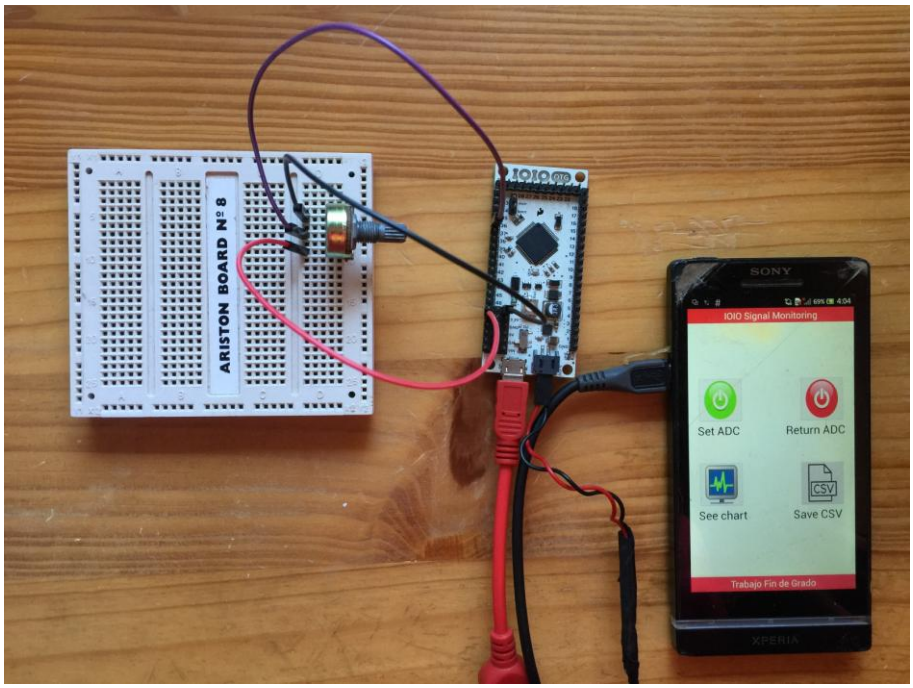


Figura 3-1. Conexionado del Sistema de Monitorización de Señales con IOIO

En el siguiente diagrama de paso de mensajes se muestra el intercambio de mensajes que tendría lugar cada vez que se programase la monitorización de una señal:

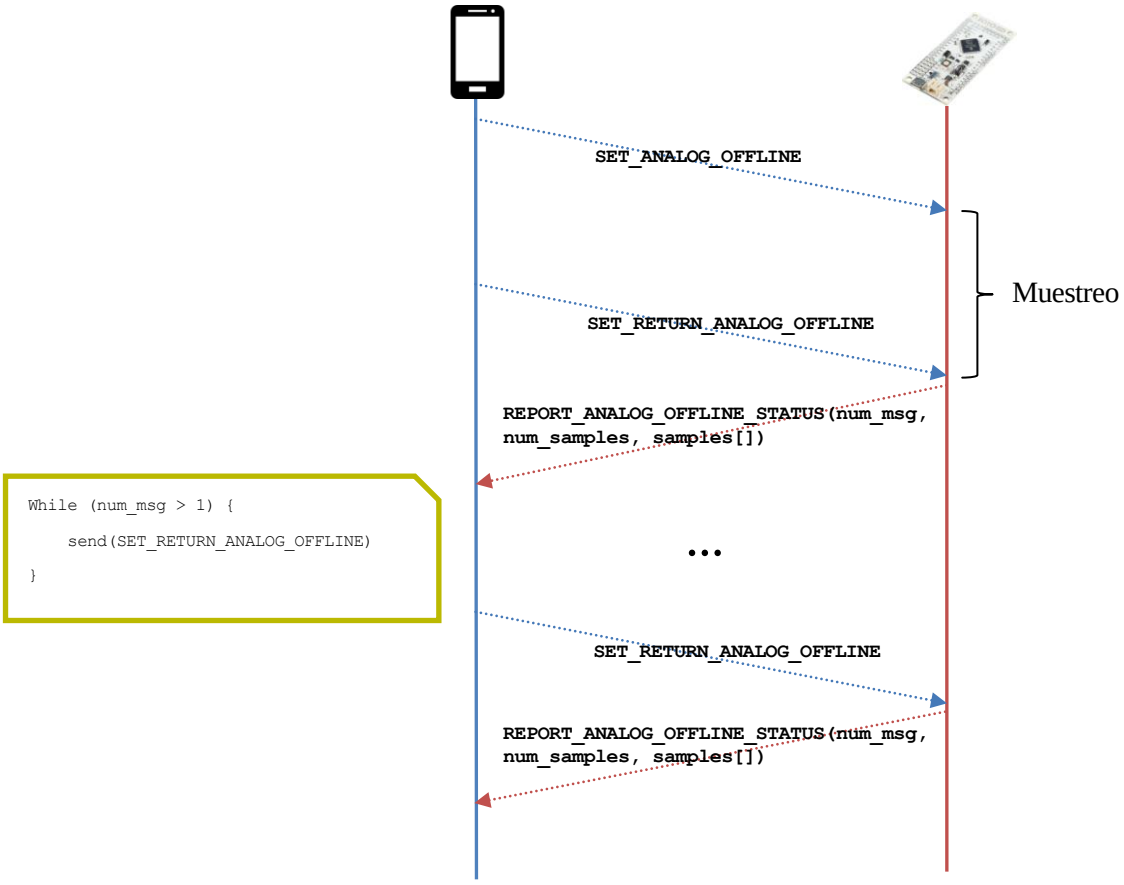


Figura 3-2. Diagrama de paso de mensajes

En primer lugar, el smartphone enviaría un mensaje `SET_ANALOG_OFFLINE` indicando a IOIO que comience una conversión A/D en el pin 36. Desde ese momento, IOIO comenzará con el muestreo de la señal e irá almacenando los resultados en su memoria RAM. Cuando el usuario desee obtenerlos, el móvil se conectará de nuevo a IOIO y enviará un mensaje `SET_RETURN_ANALOG_OFFLINE`. El micro calculará cuántos mensajes tiene que enviar en función de las muestras que tiene almacenadas y comenzará enviando un mensaje `REPORT_ANALOG_OFFLINE_STATUS` que llevará como parámetros el número de mensajes restantes (`num_msg`), el número de muestras en el mensajes (`num_samples`) y un array con las muestras (`samples[]`). El terminal comprobará si el número de mensajes restantes es mayor que uno. En caso afirmativo, se repetirá el procedimiento hasta que se hayan enviado todas las muestras.

3.2 Modificación del Firmware

3.2.1 Protocolo

Para llevar a cabo todos los cambios comentados en el apartado anterior, en primer lugar, habrá que comenzar modificando el firmware, añadiendo los nuevos tipos de mensajes. Así, en el fichero protocol_defs.h será necesario asignar un identificador a cada mensaje y definir la estructura de cada uno de ellos. Por lo que los cambios a realizar serán los siguientes:

protocol_defs.h

```
// ...
#define  NUM_SAMPLES   125
#define  NUM_MSG       10
// ...

typedef struct PACKED {
    BYTE numMsg;
    BYTE numSamples;
    WORD ADCResult[NUM_SAMPLES];
} REPORT_ANALOG_OFFLINE_STATUS_ARGS;

// ...
typedef struct PACKED {
  BYTE type;
  union PACKED {
  // ...
    REPORT_ANALOG_OFFLINE_STATUS_ARGS       report_analog_offline_status;
  } args;
} OUTGOING_MESSAGE;

typedef enum {
  // ...
  SET_ANALOG_OFFLINE                  = 0x24,
  SET_RETURN_ANALOG_OFFLINE_STATUS    = 0x25,
  REPORT_ANALOG_OFFLINE_STATUS        = 0x26,

  MESSAGE_TYPE_LIMIT
} MESSAGE_TYPE;
```

Una vez definidos los nuevos mensajes, hay que incluir en el módulo del protocolo qué se hará con los de entrada cuando estos se reciban. Además, en este mismo módulo hay que indicar el tamaño de los mensajes de salid. Así pues, en el fichero protocol.c se introducirán los siguientes cambios:

protocol.c

```
int ADCtype = 0;    // 1 if ADC Offline
  // ...
const BYTE outgoing_arg_size[MESSAGE_TYPE_LIMIT] = {
  // ...
  sizeof(REPORT_ANALOG_OFFLINE_STATUS_ARGS)
};
// ...
static BOOL MessageDone() {
  switch (rx_msg.type) {
  // ...
    case SET_ANALOG_OFFLINE:
      ADCtype=1;
      ADCOfflineSetScan();
      break;

    case SET_RETURN_ANALOG_OFFLINE_STATUS:
      ADCtype=0;
      ReportADCResults();
      break;
    // ...
  }
  return TRUE;
}
```

En el caso de recibir el mensaje `SET_ANALOG_OFFLINE`, la variable `ADCType` tomará el valor uno, indicando que se realizará una CAD offline, es decir, guardando los valores en memoria RAM. Además, se ejecutará la función `ADCOfflineSetScan()` que, como se verá más adelante, es la que inicializa el módulo de conversión analógico-digital offline.

Por el contrario, cuando se reciba el mensaje `SET_RETURN_ANALOG_OFFLINE_STATUS`, la variable `ADCType` volverá a cero y se llamará a la función `ReportADCResults()`. Esta función será la encargada de empaquetar los resultados en el mensaje `REPORT_ANALOG_OFFLINE_STATUS` y enviarlo al terminal móvil.

3.2.2 Temporizador

Para lanzar una conversión cada cierto tiempo, es necesario programar el temporizador para lanzar una interrupción que active la conversión analógico-digital. El temporizador que se ha utilizado es el TMR1. Con ese fin, se ha creado el módulo program_tmr1.{h, c}. Este módulo se compone de cuatro funciones:

- `setTMR1(ms)`: recibe como parámetro el tiempo entre interrupciones en milisegundos y se encarga de programar los registros del temporizador para generar una interrupción cada ms milisegundos, de acuerdo a:

$$T = \frac{4}{F_{osc}} \cdot Preescaler \cdot (Resolución - Precarga)$$

  Donde la frecuencia de oscilación es de 64 MHz y la resolución de 16 bits. Así pues, sólo queda establecer el registro PR1 (*Period Register*), que será con el que se comparará el TMR1 para lanzar la interrupción.

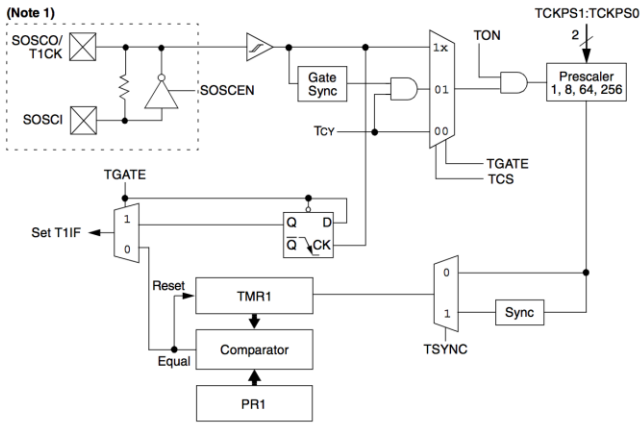


Figura 3-3. Diagrama de bloques del Timer1

Así, en función del preescaler establecido (1, 8, 64 ó 256) se tendrán cuatro diferentes intervalos de tiempo, hasta 1048 ms entre interrupciones.

- `enableTMR1()`: esta función simplemente se encarga de habilitar el temporizador poniendo a uno el bit `T1CONbits.TON`.
- `disableTMR1()`: para el temporizador poniendo el bit `T1CONbits.TON` a cero.
- `setTMR1IF(TMR1IF)`: establece el valor de la bandera de interrupción del Timer1 modificando el `bit IFS0bits.T1IF`. Se utiliza para reactivar el temporizador tras cada interrupción.

La rutina de interrupción del Timer1 se encuentra en el módulo adc_offline.{h, c}. Como se dijo anteriormente, cada vez que se tiene lugar una interrupción, se lanza una conversión analógico-digital. Así, el código de la rutina de interrupción es el siguiente:

adc_offline.c

```
// ...
void __attribute__((__interrupt__, no_auto_psv)) _T1Interrupt(void) {
    ADCOfflineTrigger();

    setTMR1IF(0);
}
```

3.2.3 Conversión Analógico-Digital

El módulo adc_offline.{h, c} es el encargado de llevar a cabo las funciones de conversión analógico-digital, guardado de resultados en memoria RAM y la transmisión de los resultados al terminal móvil.

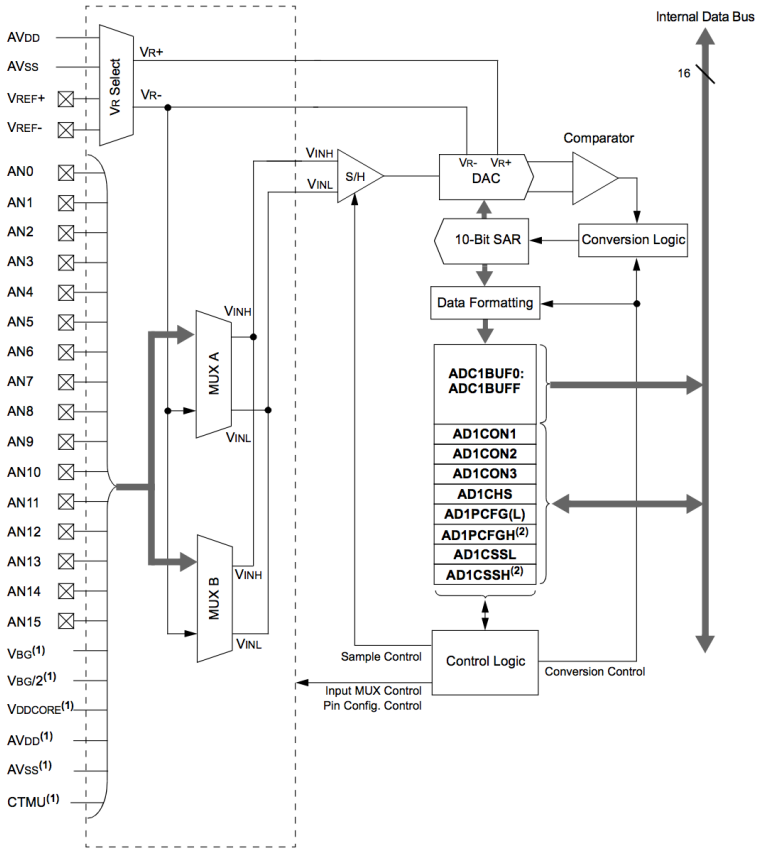


Figura 3-4. Diagrama de bloques del convertidor analógico-digital

En primer lugar, se han definido las siguientes cinco variables globales:

- `buf`: es un puntero que apunta a la dirección de ADC1BUF0, que es donde se almacenará el resultado de cada una de las conversiones A/D.
- `ADCResult[NUM_SAMPLES*NUM_MSG]`: es un vector de tamaño fijo 1250 donde se irán almacenando todas las muestras que se obtengan.
- `samplingIndex`: se inicializa a cero y se utiliza como índice para conocer el número de muestras que se han leído.
- `numPaq`: es la variable con la que se obtiene el número de paquetes que restan por enviar. Se inicializa a -1.
- `totalPaq`: indica el total de paquetes a enviar con las muestras.

Las funciones del módulo son las siguientes:

- `ADCOfflineSetScan()`: esta es la función a la que se llama cada vez que se recibe un mensaje `SET_ANALOG_OFFLINE`. Su función es la de inicializar el módulo, dando los valores adecuados a los registros para comenzar una conversión A/D en el pin 36 cada segundo.

adc_offline.c

```
void ADCOfflineSetScan() {
    ADCOfflineStop();

    AD1CON1 = 0x0000;
    AD1CON2 = 0x0000;
    AD1CON3 = 0x1F01;
    AD1CHS  = 0x0000;
    AD1CSSL = 0x0001;

    ADCOfflineStart();
}
```

Primeramente, se llama a la función `ADCOfflineStop()` para parar la conversión, por si previamente hubiera otra en marcha. Seguidamente, se han configurado los registros de control del convertidor:

- AD1CON1: se deshabilita el módulo del convertidor A/D mientras se configuran los demás registros (bit ADON 0).
- AD1CON2: establece V_{DD} y V_{SS} como tensiones de referencia, un único buffer de 16 bits y una interrupción por cada muestra.
- AD1CON3: establece el reloj del sistema como fuente, $31T_{AD}$ como tiempo de adquisición y el reloj a 8 MHz.

El siguiente registro a configurar será el de selección de canal de entrada (AD1CHS), donde se establece que se muestrea AN0 con referencia negativa.

El último registro que se configurará en esta función será el de selección de escaneo de entrada (AD1CSSL), en el cual se seleccionará el canal a escanear. En este caso, se ha seleccionado el canal 0, correspondiente al pin 36 de IOIO.

Finalmente, se procede a llamar a la función `ADCOfflineStart()`.

- `ADCOfflineStart()`: se encarga de dar comienzo con el proceso de conversión A/D, para lo cual, habilita interrupciones, inicializa variables y vectores y habilita el Timer1.

adc_offline.c

```
static inline void ADCOfflineStart() {
    _AD1IF = 0;

    _AD1IE = 1;

    int i;
    for(i=0; i<NUM_SAMPLES*NUM_MSG;i++) {
        ADCResult[i]=0;
    }
    samplingIndex=0;
    numPaq=-1;
    totalPaq=0;

    setTMR1(1000);
    enableTMR1();
}
```

En primer lugar, se baja la bandera de interrupción del convertidor A/D (`_AD1IF`), para eliminar cualquier posibilidad de interrupciones antes del comienzo. Seguidamente, se habilitan las interrupciones del convertidor (`_AD1IE`). En el siguiente paso, el vector de muestras se limpia y se inicializa a cero, así como las demás variables. Por último, se configura el Timer1 para que genere una interrupción cada 1000 ms y se habilitan las interrupciones del temporizador.

- `ADCOfflineTrigger()`: esta es la función que lanzará una conversión cada vez que se genere una interrupción del Timer1.

adc_offline.c

```
void ADCOfflineTrigger() {
    _SMPI = 0;
    _SSRC = 7;
    _CSCNA = 1;

    _ADON = 1;
    _ASAM = 1;
}
```

Antes de lanzar la conversión, se reconfiguran ciertos bits de los registros de control:

- SMPI: bits 5-2 del registro AD1CON2. Con el valor 0 se indica que habrá una interrupción por cada conversión.
- SSRC: bits 7-5 del regisro AD1CON1. Valor 7 para que empiece la conversión de forma automática.
- CSCNA: bit 10 del registro AD1CON2. Cuando vale 1 se le indica que los canales a escanear ya se programaron en el registro AD1CSSL.

Finalmente, se modifican dos bits del registro AD1CON1:

- ADON: habilita el módulo del convertidor.
- ASAM: el muestreo comienza inmediatamente después de la última conversión.

- `_ADC1Interrupt()`: la rutina de interrupción del convertidor A/D se invoca cada vez que una conversión ha finalizado. Se encuentra en el módulo de conversión "online", sobre todo por comodidad a la hora del desarrollo. Durante la rutina de interrupción, el valor de la conversión se almacenará en el vector creado para tal fin.

adc.c

```
extern int ADCtype;
extern int ADCResult[NUM_SAMPLES];
extern int samplingIndex;
// ...
void __attribute__((__interrupt__, auto_psv)) _ADC1Interrupt() {
  led_on();

  if (ADCtype==0)
    ScanDoneInterruptTrigger();
  else {
    if(samplingIndex<NUM_SAMPLES*NUM_MSG) {
        ADCResult[samplingIndex]=ADC1BUF0;
        samplingIndex++;
    }
    else if(samplingIndex==NUM_SAMPLES*NUM_MSG) {
        disableTMR1();
        ADCOfflineStop();
    }
  }
  _ADON = 0;
  _AD1IF = 0;

  int k=0;
  for(k=0; k<500;k++){}
    led_off();
}
```

La rutina de interrupción comprueba si se trata de una conversión online u offline. Si se trata de una conversión offline, almacena en el vector el valor del registro ADC1BUF0, que es el resultado de la última conversión, hasta que el vector está lleno. Si el vector se llena, se deshabilita el temporizador y la conversión. Tras esto, sse baja la bandera de interrupción y se deshabilita el módulo hasta que se vuelva a lanzar otro muestreo.

Por otro lado, se ha introducido un pequeño retardo para hacer parpadear el led de IOIO durante el muestreo y poder comprobar cuándo se está muestreando visualmente.

- `ADCOfflineStop()`: desactiva las interrupciones del temporizador y el conversor y deshabilita el módulo de conversión.

adc_offline.c

```
void ADCOfflineStop() {
    _AD1IE = 0;
    _T1IE = 0;
    _ADON = 0;
}
```

- `ReportADCResults()`: esta función es la encargada de enviar los mensajes con los resultados de la conversión desde IOIO al terminal móvil. Se le llama cada vez que se recibe el mensaje `SET_RETURN_ANALOG_OFFLINE_STATUS`.

La función comienza desactivando el temporizador y creando una variable de tipo `OUTGOING_MESSAGE`, que será el mensaje de salida `REPORT_ANALOG_OFFLINE_STATUS`.

```
void ReportADCResults() {
    int i=0;
    disableTMR1();
    SetPinDigitalOut(0, 1, 1);
    OUTGOING_MESSAGE msg;
    msg.type=REPORT_ANALOG_OFFLINE_STATUS;
// ...
```

En función del orden del mensaje a enviar habrá que distinguir tres casos distintos:

- Primer paquete.

```
// ...
    if(numPaq==-1) {
        numPaq = samplingIndex/NUM_SAMPLES;
        if (samplingIndex % NUM_SAMPLES > 0) {
            numPaq+=1;
        }
        totalPaq=numPaq;
        msg.args.report_analog_offline_status.numMsg=numPaq;

        if(samplingIndex<NUM_SAMPLES) {
            msg.args.report_analog_offline_status.numSamples=samplingIndex;
            for(i=0; i<samplingIndex;i++) {
             msg.args.report_analog_offline_status.ADCResult[i]=ADCResult[i];
            }
        }

        else {
            msg.args.report_analog_offline_status.numSamples=NUM_SAMPLES;
            for(i=0; i<NUM_SAMPLES;i++) {
             msg.args.report_analog_offline_status.ADCResult[i]=ADCResult[i];
            }
        }
    }
// ...
```

En caso de que se vaya a enviar el primer paquete, primero habrá que calcular el número de paquetes que hay que enviar. Una vez hecho esto, se incluye en el argumento del paquete `numMsg` el número de paquete al que correspondería.

Seguidamente, hay que comprobar si el número de muestras que hay es superior o inferior al tamaño del paquete. Si es inferior, el número de muestras del paquete será el de la variable `samplingIndex`, y si es superior el paquete irá completo (`NUM_SAMPLES`). En ambos casos, el vector del paquete se irá rellenando en función del número de muestras que vayan en el paquete.

- Último paquete

```
// ...
    else if (numPaq==2) {
        numPaq--;
        msg.args.report_analog_offline_status.numMsg=numPaq;

        if(samplingIndex%NUM_SAMPLES==0) {
            msg.args.report_analog_offline_status.numSamples=NUM_SAMPLES;
            for(i=(totalPaq-numPaq)*NUM_SAMPLES; i<(totalPaq-
numPaq+1)*NUM_SAMPLES;i++) {
                msg.args.report_analog_offline_status.ADCResult[i-(totalPaq-
numPaq)*NUM_SAMPLES]=ADCResult[i];
            }
        }

        else {
            msg.args.report_analog_offline_status.numSamples=samplingIndex %
NUM_SAMPLES;
            for(i=(totalPaq-numPaq)*NUM_SAMPLES; i<samplingIndex;i++) {
                msg.args.report_analog_offline_status.ADCResult[i-(totalPaq-
numPaq)*NUM_SAMPLES]=ADCResult[i];
            }
        }
    }
// ...
```

Para el último paquete se distinguen dos casos: o bien, el paquete está lleno, o bien, el número de muestras es menor al tamaño del paquete.

- Paquete intermedio

```
// ...
    else {
        numPaq--;
        msg.args.report_analog_offline_status.numMsg=numPaq;
        msg.args.report_analog_offline_status.numSamples=NUM_SAMPLES;
        for(i=(totalPaq-numPaq)*NUM_SAMPLES; i<(totalPaq-
numPaq+1)*NUM_SAMPLES;i++) {
            msg.args.report_analog_offline_status.ADCResult[i-(totalPaq-
numPaq)*NUM_SAMPLES]=ADCResult[i];
        }
    }
// ...
```

Los paquetes intermedios van llenos, por lo que directamente se rellenan con todas las muestras en función del número de paquete.

Finalmente, se envía el mensaje mediante la función `AppProtocolSendMessage`:

```
// ...
    AppProtocolSendMessage(&msg);
}
```

3.2.4 Características generales

El módulo features.{h, c}, como se dijo anteriormente, es el que contiene las funciones generales, entre ellas las de reseteo. Cada vez que un terminal se desconecta, automáticamente IOIO ejecuta un soft reset o "reinicio blando". Este reinicio hace que se ejecuten funciones que reinician los pines y los establece todos como entradas digitales. Por tanto, se han realizado una serie de modificaciones para lograr que el microcontrolador pueda seguir muestreando una señal a pesar de no tener el terminal móvil conectado.

En primer lugar, habrá que saber cuándo se está en modo ADC offline, por lo que será necesario definir la variable global `ADCType` en features.c:

features.c

```
// ...
extern int ADCtype;
// ...
```

Cuando la variable `ADCType` tome el valor cero, se estará en modo online, y será entonces cuando cada vez que se desconecte el dispositivo Android, se pueda reiniciar el módulo del convertidor:

```
// ...
void SoftReset() {
  PRIORITY(7) {
    log_printf("SoftReset()");
    TimersInit();
    PinsInit();
    PWMInit();
    if(ADCtype==0)
        ADCInit();
    UARTInit();
    SPIInit();
    I2CInit();
    InCapInit();
    SequencerInit();
  }
}
// ...
```

Por último, quedará modificar la función `PinsInit()`. Habrá que modificarla de tal forma que reinicie todos los pines a excepción del pin 36, por el que se continuará monitorizando una señal:

```
// ...
static void PinsInit() {
  int i;
  _CNIE = 0;
  SetPinDigitalOut(0, 1, 1);

  for (i = 1; i < NUM_PINS; ++i) {
      if((ADCtype==1 && i!=36 && i!=40) || ADCtype==0)
        SetPinDigitalIn(i, 0);
  }

  _CNIF = 0;
  _CNIE = 1;
  _CNIP = 1;
}
// ...
```

3.3 Modificación de la librería Android

Los cambios realizados en el firmware de IOIO hay que reflejarlos en la parte de la librería de Android. El lenguaje utilizado para la programación de la librería es Java, por tanto, el módulo del protocolo es una clase. La clase `IOIOProtocol` contiene todas las variables y métodos necesarios para llevar a cabo la comunicación entre el teléfono y IOIO. Así, haciendo uso de las clases `InputStream`, `OutputStream` y `IncomingHandler`, es posible escribir o leer datos de IOIO.

De esta forma, en primer lugar, hay que comenzar añadiendiendo los mensajes creados al módulo del protocolo en la librería:

IOIOProtocol.java

```
class IOIOProtocol {
      static final int HARD_RESET                          = 0x00;
      // ...
      static final int SET_ANALOG_OFFLINE                  = 0x24;
      static final int SET_RETURN_ANALOG_OFFLINE_STATUS    = 0x25;
      static final int REPORT_ANALOG_OFFLINE_STATUS        = 0x26;
// ...
```

3.3.1 Mensajes de salida

Dentro de la clase `IOIOProtocol` se definen los métodos para escribir los mensajes de salida, es decir, los mensajes que van desde Android al microcontrolador. Por tanto, habrá que definir dos nuevos métodos para enviar los mensajes `SET_ANALOG_OFFLINE` y `SET_RETURN_ANALOG_OFFLINE_STATUS`.

IOIOProtocol.java

```
// ...
    synchronized public void setAnalogOffline() throws IOException {
            beginBatch();
            writeByte(SET_ANALOG_OFFLINE);
            endBatch();
      }

      synchronized public void setReturnAnalogOfflineStatus() throws
IOException {
            beginBatch();
            writeByte(SET_RETURN_ANALOG_OFFLINE_STATUS);
            endBatch();
      }
// ...
```

Básicamente, cada uno de estos métodos envía a IOIO un byte con el código del mensaje utilizando el método `writeByte`, que escribe un byte en el flujo de salida. Los métodos `beginBatch` y `endBatch` optimizan la operación. Por otro lado, hay que señalar que los métodos utilizan la palabra reservada `synchronized` debido a que se trata con diferentes hilos y, de esta forma, se protege el acceso concurrente a variables y objetos.

Una vez que se tienen los métodos del protocolo, hay que añadir dos métodos nuevos en la interfaz `IOIO` que, posteriormente, serán los que se usarán en la aplicación Android. Así, se definen los siguientes métodos en la interfaz:

IOIO.java

```
public interface IOIO {
// ...
   public void setAnalogOffline() throws ConnectionLostException;
   public void setReturnAnalogOfflineStatus() throws ConnectionLostException;
// ...
}
```

El siguiente paso es implementar los métodos en la clase `IOIOImpl`:

IOIOImpl.java

```
public class IOIOImpl implements IOIO, DisconnectListener {
// ...
      @Override
      public void setAnalogOffline() throws ConnectionLostException {
            try {
                  protocol_.setAnalogOffline();
            } catch (IOException e) {
                  throw new ConnectionLostException(e);
            }
      }

      @Override
      public void setReturnAnalogOfflineStatus() throws
ConnectionLostException {
            try {
                  protocol_.setReturnAnalogOfflineStatus();
            } catch (IOException e) {
                  throw new ConnectionLostException(e);
            }
      }
}
```

En cada uno de los métodos, se utiliza una instancia de la clase `IOIOProtocol` que invoca al método correspondiente de los creados anteriormente.

3.3.2 Mensajes de entrada

Para desarrollar la parte en la que intervienen los mensajes de entrada, en primer lugar, hay que comenzar definiendo el método "manejador" del servicio. Para ello, se define el método handleAnalogOffline en la interfaz IncomingHandler:

IOIOProtocol.java

```
class IOIOProtocol {
// ...
      public interface IncomingHandler {
      // ...
            public void handleAnalogOffline(int[] ADCResult, int numMsg);
      }
// ...
```

La implementación de la interfaz se lleva a cabo en la clase IncomingState, donde simplemente se asignará el valor de las variables del método a las variables de la clase:

IncomingState.java

```
class IncomingState implements IncomingHandler {
// ...
      public int[] ADCResult_;
      public int numMsg_;
// ...
      @Override
      public void handleAnalogOffline(int[] ADCResult, int numMsg) {
            ADCResult_=ADCResult;
            numMsg_=numMsg;
      }
// ...
```

Seguidamente, es necesario añadir al hilo de lectura de mensajes de entradas el caso de que se reciba el mensaje `REPORT_ANALOG_OFFLINE_STATUS`, donde habrá que leer las muestras:

IOIOProtocol.java

```
class IOIOProtocol {
// ...
      class IncomingThread extends Thread {
// ...
            @Override
            public void run() {
                  super.run();
// ...
                  int arg1;
                  int arg2;
// ...
                  try {
                        while (true) {
                              switch (arg1 = readByte()) {
// ...
                              case REPORT_ANALOG_OFFLINE_STATUS:
                                    int numMsg=readByte();
                                    int numSamples=readByte();
                                    int[] ADCResult=new int[numSamples];
                                    for(int i=0; i<AnalogMsgLength;i++){
                                          if (i < numSamples) {
                                                arg1=readByte();
                                                arg2=readByte();
                                                ADCResult[i]=arg1 | (arg2<<8);
                                          }
                                          else {
                                                arg1=readByte();
                                                arg2=readByte();
                                          }
                                    }
                                    handler_.handleAnalogOffline(ADCResult,
numMsg);
                              break;

// ...
```

Una vez que el terminal recibe el mensaje `REPORT_ANALOG_OFFLINE_STATUS`, en segundo lugar lee el número de mensajes (`numMsg`), que corresponde al primer byte. El siguiente byte corresponde al número de muestras del mensaje y se almacena en la variable `numSamples`. Una vez se conoce el número de muestras se crea el vector de tamaño `numSamples` y se comienza a leer el mensaje completo muestra por muestra.

Como el tamaño de la muestra es de 10 bits, será necesario leer dos bytes y reconstruirla, para posteriormente almacenarla en el vector `ADCResult`. Si el mensaje no está lleno de muestras, habrá que leer el mensaje hasta el final para no tener errores. Finalmente, los resultados se envían al manejador (`handler_`).

Posteriormente, será necesario definir en la interfaz de IOIO los métodos que se podrán utilizar en la aplicación Android. En este caso serán dos: uno para obtener las muestras y otro para obtener el número de mensajes.

IOIO.java

```
public interface IOIO {
// ...
    public int[] getADCResult() throws ConnectionLostException;
    public int getNumMsg() throws ConnectionLostException;
// ...
}
```

Finalmente, al igual que se hizo con los mensajes de salida, hay que implementar los métodos definidos en la interfaz.

IOIOImpl.java

```
public class IOIOImpl implements IOIO, DisconnectListener {
// ...
      @Override
      public int[] getADCResult() throws ConnectionLostException {
            try{
                  return incomingState_.ADCResult_;
            } finally {

            }
      }

      @Override
      public int getNumMsg() throws ConnectionLostException {
            try{
                  return incomingState_.numMsg_;
            } finally {

            }
      }
// ...
```

Como se puede ver, la implementación devuelve la variable `numMsg_` o `ADCResult_`, según corresponda, del objeto `incomingState_`, donde se almacenaron los resultados anteriormente.

4 APLICACIÓN ANDROID

El día que la mierda tenga algún valor, los pobres nacerán sin culo.

- Gabriel García Márquez -

Para comenzar el desarrollo de la aplicación Android, en primer lugar será necesaria la importación de la librería desarrollada en el apartado anterior, que ya incluiría las nuevas funciones de las que se harán uso en la aplicación. Una vez se ha importado el fichero JAR de la librería e incluido en Android Studio, ya es posible utilizar las clases y métodos de IOIO.

Antes de empezar, es necesario incluir en el manifiesto los permisos de Internet y Bluetooth para la comunicación de la aplicación con IOIO. Además, como los resultados de la conversión analógico-digital se almacenarán en memoria en un fichero CSV, será necesario asignar los permisos para que la aplicación pueda escribir en memoria:

AndroidManifest.xml

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

La aplicación constará de dos actividades:

- `MainActivity`: el layout o diseño de esta actividad constará de cuatro botones, cada uno encargado de una función diferente:
    - *Set ADC*: dará la orden de comienzo de monitorización. Tras pulsar el botón IOIO comenzará a muestrear una señal por el pin 36 y el móvil podrá ser desconectado hasta que el usuario quiera los resultados.
    - *Stop ADC*: parará la monitorización de la señal y obtendrá los datos de vuelta.
    - *Save CSV*: almacenará en memoria un fichero CSV con los valores de las muestras.
    - *See chart*: conducirá a la segunda actividad, donde se muestra una gráfica con la evolución de la señal respecto al tiempo.

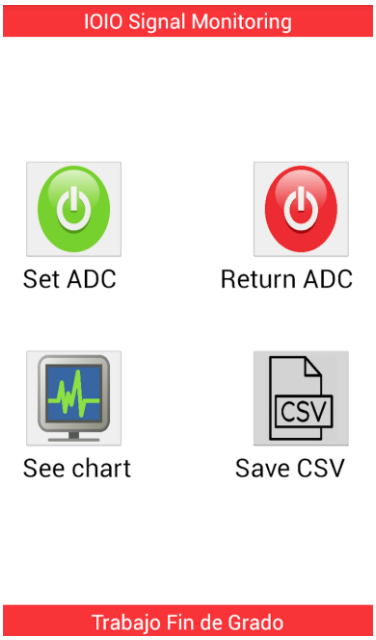


Figura 4-1. Actividad `MainActivity`

- `SeeChartActivity`: esta actividad es accesible a través del botón See chart y mostrará los resultados en una gráfica.

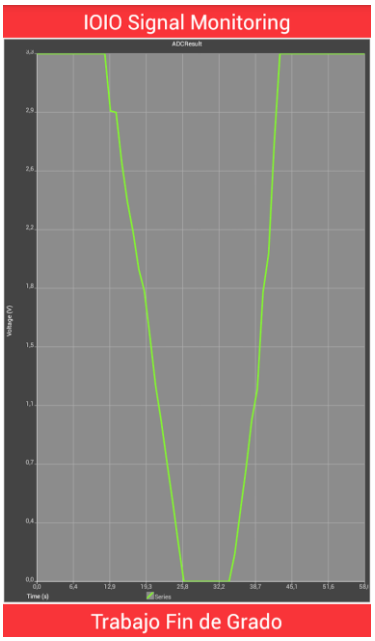


Figura 4-2. Actividad `SeeChartActivity`

En los siguientes apartados se detallará el desarrollo de cada una de las actividades y de las variables y métodos que lo forman.

4.1 Actividad MainActivity

La actividad `MainActivity` hereda de `IOIOActivity` para implementar los objetos y métodos que permiten el control de IOIO. Las variables de instancia que se han definido han sido las siguientes:

- `myioio`: es un objeto de la clase `IOIO` que servirá para obtener una copia del objeto `ioio_` que nos permita acceder a él fuera de la clase `Looper`.
- msgResultList: lista enlazada que recoge los resultados de la conversión incluidos en cada mensaje.
- ADCResult: vector que recoge todas las muestras obtenidas tras una monitorización.
- dateSet: objeto de tipo Date para obtener la fecha y hora en la que empieza la monitorización.

MainActivity.java

```
public class MainActivity extends IOIOActivity {
    // ...
    private IOIO myioio;
    private List<int[]> msgResultList = new LinkedList<>();
    private int[] ADCResult;
    private Date dateSet;
// ...
```

Una vez que se establece conexión con IOIO, lo primero que se hará es copiar el objeto `ioio_` al objeto `myioio`, con el fin de poder acceder a él fuera de la clase `Looper`. Asimismo, cuando el terminal se conecte a IOIO se utilizará el método `enableUi`, que habilita los botones de la interfaz.

```
 // ...
    class Looper extends BaseIOIOLooper {

        @Override
        protected void setup() throws ConnectionLostException {
            myioio = ioio_;
            enableUi(true);
        }

        @Override
        public void loop() throws ConnectionLostException {

        }

        @Override
        public void disconnected() {
            enableUi(false);
        }
}
// ...
```

Tras establecer la conexión con IOIO, hay que programar la monitorización de la señal. Como se vio anteriormente, esto empieza con el envío del mensaje `SET_ANALOG_OFFLINE`. Para ello, en la librería se creó el método `setAnalogOffline`. Así, se ha creado en Android el método `setADCOffline`, al que se llama cada vez que se pulsa el botón *Set ADC*.

```
// ...
    public void setADCOffline(View view) throws ConnectionLostException,
InterruptedException {
        myioio.setAnalogOffline();
        myioio.disconnect();
        dateSet = Calendar.getInstance().getTime();
    }
// ...
```

Después de ejecutar el método `setAnalogOffline`, se realiza la desconexión con IOIO invocando al método `disconnect`. Además, en este momento se guarda la fecha y hora de comienzo de muestreo en la variable `dateSet`, que posteriormente servirá para reflejarla en la gráfica y en el fichero CSV.

Por otro lado, para que al pulsar el botón *Set ADC* se ejecute el método `setADCOffline`, hay que especificar en el layout del botón el método que se ejecutará en la opción `android:onClick`. Lo mismo se hará con cada uno de los botones y sus respectivos métodos.

activity_main.xml

```
<!-- ... -->
    <ImageButton
            android:id="@+id/btnSetADC"
            android:layout_width="100dp"
            android:layout_height="100dp"
            android:adjustViewBounds="true"
            android:scaleType="fitXY"
            android:text="Start monitoring"
            android:paddingLeft="15dip"
            android:paddingRight="15dip"
            android:src="@drawable/startbutton"
            android:clickable="false"
            android:onClick="setADCOffline"
            android:layout_alignLeft="@+id/textView"
            android:layout_alignStart="@+id/textView" />
<!-- ... -->
```

El siguiente paso sería pulsar el botón *Return ADC* para obtener las muestras guardadas en la memoria RAM de IOIO. Para ello se ha desarrollado el método `returnADCResult`.

```
// ...
    public void returnADCResult(View view) throws ConnectionLostException,
InterruptedException {
        int[] msgResult;
        int numMsg = 0;
        while (numMsg != 1) {
            myioio.setReturnAnalogOfflineStatus();
            Thread.sleep(300);
            numMsg = myioio.getNumMsg();
            Thread.sleep(300);
            Log.v(TAG, "numMsg= " + numMsg);
            msgResult = myioio.getADCResult();
            Thread.sleep(300);
            msgResultList.add(myioio.getADCResult());
        }
    // ...
```

Lo primero que realiza el método es enviar el mensaje `SET_RETURN_ANALOG_OFFLINE`, invocando al método `setReturnAnalogOfflineStatus`. Tras esto, se deja un invervalo de 300 ms para esperar la respuesta del primer mensaje y obtener, mediante el método `getNumMsg`, el número de mensajes restantes para obtener todas las muestras. Seguidamente se obtienen los resultados del mensajes y se añaden a la lista enlazada. Esto se realizará varias veces hasta que se reciba el último mensaje.

Tras obtener todas las muestras, hay que reorganizarlas e incluirlas en el mismo array ADCResult:

```
        // ...
        int arrayLength = 0;
        for (int i = 0; i < msgResultList.size(); i++) {
            msgResult = msgResultList.get(i);
            arrayLength += msgResult.length;
        }
        ADCResult = new int[arrayLength];
        int samples = 0;
        for (int i = 0; i < msgResultList.size(); i++) {
            msgResult = msgResultList.get(i);
            for (int j = 0; j < msgResult.length && samples <
ADCResult.length; j++) {
                ADCResult[samples] = msgResult[j];
                samples++;
            }
        }
        Log.v(TAG, "ADCResult=" + Arrays.toString(ADCResult));
        alertResult();
        //setText(Arrays.toString(ADCResult));
        myioio.disconnect();
    }
// ...
```

Una vez que están todas las muestras reorganizadas, se llama al método `alertResult`, que mostrará la siguiente alerta en pantalla:

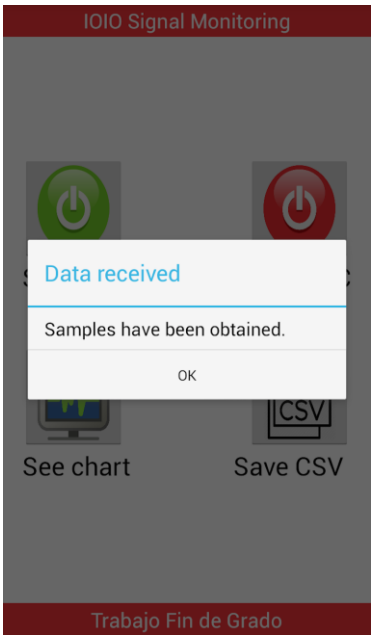


Figura 4-3. Alerta tras recibir las muestras

A partir de este momento se tienen dos posibilidades: o bien se visualizan los datos en pantalla gráficamente, o bien se almacenan los datos en la memoria del teléfono en un fichero CSV. Si se elige esta última opción, el método que se ejecutará será `saveCSV`:

```
// ...
    public void saveCSV(View view) throws ConnectionLostException,
InterruptedException {
        File folder = new File(Environment.getExternalStorageDirectory() +
                File.separator + "IOIO");
        boolean success = true;
        if (!folder.exists()) {
            success = folder.mkdirs();
        }
        if (success) {
            SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddHHmmss");
            File file = new File(Environment.getExternalStorageDirectory() +
                    File.separator + "IOIO" + File.separator + "Samples" +
sdf.format(dateSet) + ".csv");
            if (!file.exists()) {
                try {
                    file.createNewFile();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                try {
                    FileOutputStream fos = new FileOutputStream(file);

                    String sampleToWrite;
                    for (int i=0; i<ADCResult.length; i++) {
                        sampleToWrite = Integer.toString(ADCResult[i]);
                        if ((i + 1) < ADCResult.length) {
                            sampleToWrite += ",";
                        }
                        fos.write(sampleToWrite.getBytes());
                    }
                    alertCSV(file.getPath());
                    fos.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        } else {
            // ...
        }
    }
// ...
```

El método crea el directorio IOIO en el terminal si no existe previamente. Tras ello, crea dentro de ese directorio un fichero de nombre “Samplesfecha.csv” y comienza a escribir una muestra tras otra separadas por coma. Las muestras se almacenarán tal y como son recibidas, de forma que no se pierdan datos por el redondeo al convertir el valor a `float`. Cuando termina se genera la siguiente alerta en pantalla:

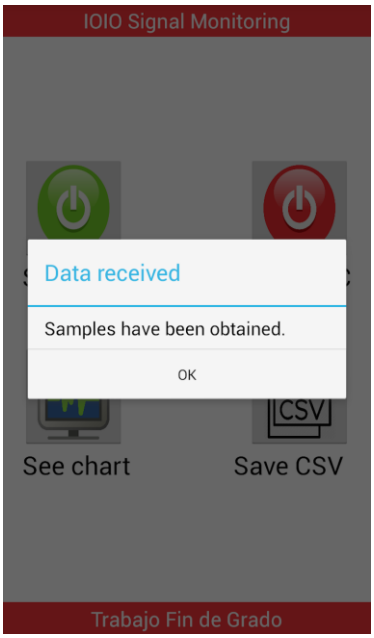


Figura 4-4. Alerta tras crear el fichero CSV

Conectando el móvil al ordenador y accediendo a su memoria interna se puede ver el fichero CSV:

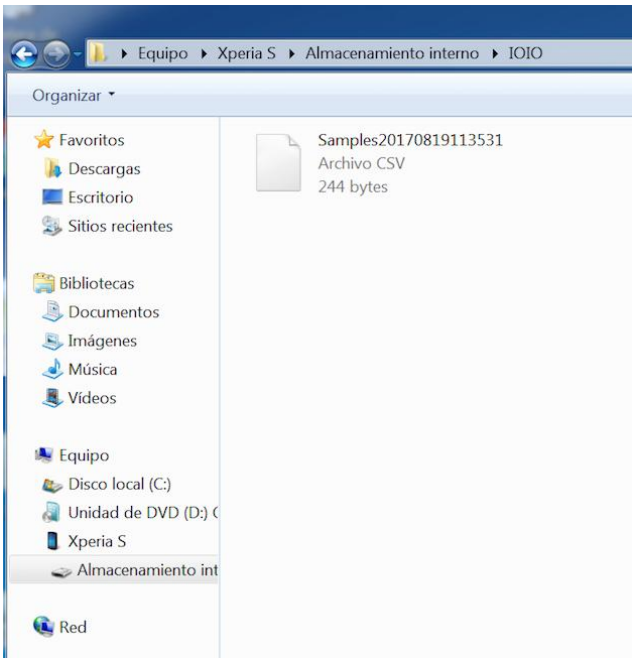


Figura 4-5. Directorio y fichero con las muestras

El fichero tendrá la siguiente forma:

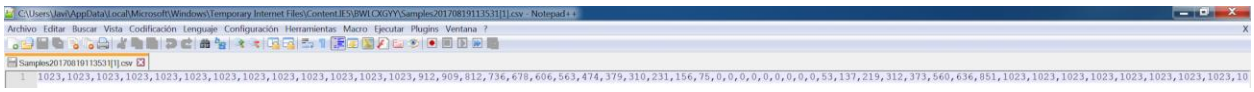


Figura 4-6. Fichero CSV

Por último, como se dijo anteriormente, los datos se pueden observar en una gráfica pulsando el botón *See chart*. En ese caso se lanzará una nueva actividad, por lo que el método `seeChart` solo lanzará la actividad `SeeChartActivity` y le enviará la variable `ADCResult`.

```
// ...
    public void seeChart(View view) throws ConnectionLostException,
InterruptedException {
        Intent intent = new Intent(this, SeeChartActivity.class);
        intent.putExtra("ADCResult", ADCResult);
        startActivity(intent);
    }
// ...
```

4.2 Actividad SeeChartActivity

Para representar gráficamente los resultados se ha utilizado la librería androidplot [18]. Esta librería proporciona las clases, métodos y diseños necesarios para tal fin. En primer lugar, se ha de añadir al layout la gráfica vacía de contenido, sólo con los títulos.

activity_see_chart.xml

```
<!-- ... -->
    <com.androidplot.xy.XYPlot
        android:id="@+id/plot"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        androidplot.domainLabel="Time (s)"
        androidplot.rangeLabel="Voltage (V)"
        androidplot.title="ADCResult"/>
<!-- ... -->
```

La actividad en este caso será simple, puesto que los datos ya han sido obtenidos y sólo queda dibujar la gráfica. Así, sólo será necesario el método onCreate, ya que la actividad se creará con la gráfica ya dibujada.

SeeChartActivity.java

```
public class SeeChartActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_see_chart);

        XYPlot plot;
        plot = (XYPlot) findViewById(R.id.plot);
        int[] ADCResult;
        ADCResult=getIntent().getExtras().getIntArray("ADCResult");
        float [] VoltageADC = new float[ADCResult.length];
        for (int i=0; i<ADCResult.length; i++) {
            VoltageADC[i]=ADCResult[i];
        }
        Number[] seriesNumbers=new Number[ADCResult.length];
        for(int i=0; i<ADCResult.length; i++) {
            seriesNumbers[i]=VoltageADC[i]/1023*3.3;
        }

        XYSeries series = new SimpleXYSeries(Arrays.asList(seriesNumbers),
SimpleXYSeries.ArrayFormat.Y_VALS_ONLY, "Series");
        LineAndPointFormatter seriesFormat = new
LineAndPointFormatter(Color.rgb(127, 255, 0), 0x000000, 0x000000, null);
        plot.addSeries(series, seriesFormat);
    }
}
```

El objeto `plot` es el que representa a la gráfica. Una vez creado, se obtienen los datos del `Intent` y se convierten a valores de tensión. Estos se almacenarán en el objeto `seriesNumbers`, de tipo `Number`.

El objeto series es el que representa la línea gráfica. Como las muestras están separadas por intervalos de un segundo, no será necesario dar valores al eje X de la gráfica, ya que automáticamente las separa por espacio de una unidad. Finalmente, se da formato a la gráfica con el objeto `seriesFormat` y, mediante el método `addSeries`, se añade la línea gráfica. De esta forma, tras una lectura, se tendrá una gráfica de este estilo:

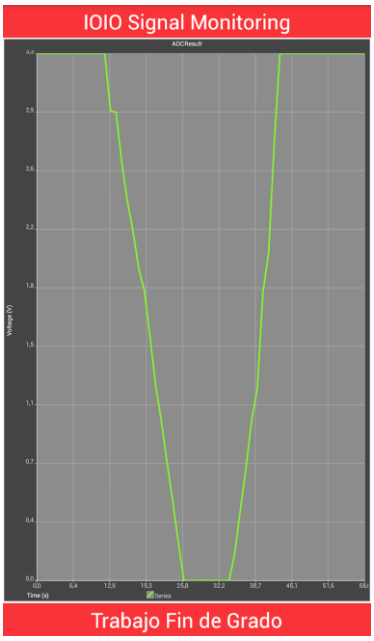


Figura 4-7. Gráfica con la evolución de la señal en función del tiempo

5 PRUEBAS Y VALIDACIÓN

Un científico debe tomarse la libertad de plantear cualquier cuestión, de dudar de cualquier afirmación, de corregir errores.

- Robert Oppenheimer -

Las pruebas [19] [20] constituyen una parte vital en el desarrollo del software. Con frecuencia, la prueba requiere más esfuerzo que cualquier otra acción del desarrollo. Por tanto, esto requiere un orden a la hora de probar el software para no desperdiciar tiempo y esfuerzo o pasar algunos errores desapercibidos.

Las primeras etapas de pruebas se enfocan sobre un solo componente o pequeños grupos de componentes relacionados, con el fin de descubrir errores en los datos y en la lógica de procesamiento que se encapsularon en los componentes. Tras esto, debe integrarse hasta que se construya el sistema completo, cuando se efectuarán las pruebas de integración.

La prueba de software es un elemento de un tema más amplio que usualmente se conoce como verificación y validación (V&V). La *verificación* se refiere al conjunto de tareas que garantizan que el software implementa correctamente una función específica. La *validación* es un conjunto diferente de tareas que aseguran que el software que se construye sigue los requisitos establecidos.

En este capítulo se va a ahondar en cada una de las pruebas realizadas hasta completar todo el desarrollo del software y su validación.

5.1 Pruebas parciales

Las pruebas parciales están destinadas a probar cada "unidad" (clase, función, componente, módulo, …). De esta forma, se verifica si el diseño y la programación es correcta. El diseño de estas pruebas puede ser previo a la codificación o ir unida a esta.

Al no ser un componente un programa independiente, se necesita desarrollar software de sobrecarga, es decir, recursos necesarios para probar el módulo. Por ejemplo, una función Main, recoger algunos datos de entrada, etc.

5.1.1 Timer

En primer lugar, para el desarrollo de un sistema de monitorización de señales es necesario controlar todo lo relativo a los temporizadores del microcontrolador. Para ello, se ha utilizado un firmware básico en el microcontrolador a partir del cual se han introducido una serie de modificaciones para llevar a cabo la prueba.

Así, se ha programado el parpadeo del led interno de IOIO mediante el uso de temporizadores con su respectiva rutina de interrupción. De esta forma, cada vez que expira el temporizador, la rutina de interrupción comprueba el estado del led. Si estaba apagado lo enciende, y si estaba encendido lo apaga.

El código de la prueba sería el siguiente:

main.c

```
#include "Compiler.h"
#include "timer.h"
#include "platform.h"

int main() {
    led_init();
    led_on();

    T1CON = 0x0030;             // Ajusta el preescaler 1:256
    TMR1 = 0x00;                // Limpia el registro
    PR1 = 0xFFFF;               // Registo de precarga
    IPC0bits.T1IP = 0x01;       // Nivel de prioridad

    IFS0bits.T1IF = 0;          // Bandera de interrupción
    IEC0bits.T1IE = 1;          // Se habilita la interrupción del Timer1
    T1CONbits.TON = 1;          // Se activa el temporizador

    while(1);

    return 0;
}

void __attribute__((__interrupt__, no_auto_psv)) _T1Interrupt(void) {
    if (_LATC12==0) // Led encendido
        led_off();
    else
        led_on();

    IFS0bits.T1IF=0;
}
```

Como se puede apreciar en el código anterior, en la función principal (`main`) se establecen los ajustes del temporizador cargando en los registros los valores deseados: preescaler 1:256, precarga al máximo y habilitación de las interrupciones y el temporizador.

En la rutina de interrupción (`_T1Interrupt`), se comprueba el estado del led y se actúa en consecuencia, y además se baja la bandera de interrupción para comenzar de nuevo la cuenta del temporizador.

El siguiente paso, sería crear una función que reciba como argumento el tiempo entre interrupciones del temporizador y que en función de esto configure los registros del temporizador. Como se sabe que el tiempo entre interrupciones viene dado por:

$$T = \frac{4}{F_{osc}} \cdot Preescaler \cdot (Resolución - Precarga) \qquad [1]$$

siendo F_{osc} en IOIO de 64 MHz, queda establecer qué registros configurar. En función del preescaler establecido, el tiempo entre interrupciones será mayor o menor. Así, se establece un límite para cada preescaler y en función del preescaler se asigna al registro de precarga el valor que nos salga de despejar de la ecuación anterior. De esta forma, la función es la siguiente:

program_tmr1.c

```
void setTMR1(unsigned int ms) {
    TMR1 = 0x0000;
    if (ms < 4) {
        // 00 = 1:1 prescaler
        T1CON = 0x0000;
        PR1 = (unsigned int)(((double)ms*64000)/4);
    }
    if(ms > 4 && ms < 32) {
        // 01 = 1:8 prescaler
        T1CON = 0x0010;
        PR1 = (unsigned int)(((double)ms*64000)/(4*8));
    }
    if(ms > 33 && ms < 262) {
        // 10 = 1:64 prescaler
        T1CON = 0x0020;
        PR1 = (unsigned int)(((double)ms*64000)/(4*64));
    }
    if(ms > 263) {
        // Hasta 1048 ms
        // 11 = 1:256 prescaler
        T1CON = 0x0030;
        PR1 = (unsigned int)(((double)ms*64000)/(4*256));
    }

    IPC0bits.T1IP = 0x01;   // Prioridad de la interrupción
    IFS0bits.T1IF = 0;      // Bandera de interrupción
    IEC0bits.T1IE = 1;      // Activamos las interrupciones
    T1CONbits.TON = 1;      // Activamos el Timer1
}
```

Para comprobar que los resultados son correctos es necesario un osciloscopio para medir la señal y comprobar que el periodo en el que se mantiene la salida digital a uno, y luego a cero, es el que se ha configurado.

5.1.2 Mensajes

Para probar el funcionamiento del paso de mensajes entre el dispositivo y el firmware de IOIO se realizarán una serie de modificaciones con el fin de estudiar cómo funciona el paso de mensajes entre Android y IOIO.

Las pruebas a ejecutar serán:

1) Modificación del mensaje `SET_PIN_DIGITAL_OUT` para programar el parpadeo del led STAT mediante la rutina de interrupción del TMR1.
2) Creación de un nuevo mensaje `SET_BLINK_TMR1`, que realiza las mismas funciones que el anterior.
3) Creación del mensaje `SET_PIN_BLINK_TMR1`, que realiza las mismas funciones que los mensjaes anteriores, pero que tiene como parámetro el tiempo entre interrupciones. Además, se ha creado otro mensaje (`SET_BLINK_TMR1`), que activa o desactiva el Timer1 en función del parámetro enable. Al desactivar el temporizador, también apaga el led si estaba encendido.
4) Modificación del mensaje `SET_PIN_BLINK_TMR1` para que también incluya como parámetro el pin que se desea poner como salida digital.
5) Creación de un mensaje de salida desde el microcontrolador hacia el terminal.

5.1.2.1 Modificación de un mensaje

Con esta prueba se quiere comprobar que los eventos del temporizador funcionan correctamente. Para ello, cada vez que el pin 0 se seleccione como salida digital se va a programar el Timer1 con el fin de que este led se quede parpadeando. Si todo es correcto, una vez que se desconecte IOIO del terminal, el led debe seguir de la misma forma hasta que se vuelva a conectar, cuando tendrá lugar un *soft reset* y el pin volverá a definirse como entrada digital.

Para llevar a cabo esta prueba es necesario modificar el módulo protocol.{h, c} que se encarga de analizar todos los mensajes del protocolo. Así, cuando el mensaje recibido sea `SET_PIN_DIGITAL_OUT`, si el pin seleccionado es el 0 (pin del led STAT), se configurará el Timer1 para generar una interrupción cada 200 ms que hará que el led parpadee. Mediante el mensaje `SET_DIGITAL_OUT_LEVEL` se activará y desactivará el temporizador.

En el código del firmware esto se programa de la siguiente forma:

protocol.c

```
// ...
case SET_PIN_DIGITAL_OUT:
    CHECK(rx_msg.args.set_pin_digital_out.pin < NUM_PINS);
    if(rx_msg.args.set_pin_digital_out.pin == 0)
        setTMR1(200);

    SetPinDigitalOut(rx_msg.args.set_pin_digital_out.pin,
                rx_msg.args.set_pin_digital_out.value,
                rx_msg.args.set_pin_digital_out.open_drain);
break;

case SET_DIGITAL_OUT_LEVEL:
    CHECK(rx_msg.args.set_digital_out_level.pin < NUM_PINS);
    if(rx_msg.args.set_digital_out_level.pin == 0 &&
rx_msg.args.set_digital_out_level.value == 0)
        enableTMR1();
    if(rx_msg.args.set_digital_out_level.pin == 0 &&
rx_msg.args.set_digital_out_level.value == 1) {
        stopTMR1();
        SetDigitalOutLevel(rx_msg.args.set_digital_out_level.pin,
                    rx_msg.args.set_digital_out_level.value);
    }
    else
        SetDigitalOutLevel(rx_msg.args.set_digital_out_level.pin,
                  rx_msg.args.set_digital_out_level.value);
break;
// ...
```

Será en la rutina de interrupción del Timer1 donde se programará el parpadeo del led utilizando las funciones ya disponibles para ello en el fichero pins.h:

main.c

```
// ...
void __attribute__((__interrupt__, no_auto_psv)) _T1Interrupt(void) {

    if(value==0) {
        value = 1;
        PinSetLat(0, value);
    }
    else {
        value = 0;
        PinSetLat(0, value);
    }

    setTMR1IF(0);
}
```

Así, se comprueba que, efectivamente, el led sigue parpadeando una vez que se desconecta IOIO de Android y, por tanto, se ha programado correctamente el Timer1.

5.1.2.2 Creación de un mensaje de entrada

En esta prueba se va a crear un nuevo tipo de mensaje de entrada a IOIO (`SET_BLINK_TMR1`), mediante el cual se va a activar el Timer1 y se conseguirá lo mismo que en las anteriores pruebas: que el led interno de IOIO parpadee. El objetivo de esta prueba es comprobar y asegurar que tanto en el firmware de IOIO como en la librería de Android se programa correctamente todo lo relativo a los mensajes de entrada a IOIO.

Al crear un nuevo tipo de mensaje, este debe ser interpretado por ambos lados de la comunicación. Por tanto, será necesario modificar tanto el firmware del microcontrolador como la librería de Android.

5.1.2.2.1 Firmware

Todos los mensajes están identificados dentro del firmware en el fichero protocol_defs.h. Ahí se definen tanto el tipo de mensaje como los parámetros de estos. De esta forma, como el mensaje creado sólo da la orden de configuración del timer y lo activa, únicamente es necesario definir el tipo de mensaje, añadiéndolo a los ya existentes:

protocol_defs.h

```
// ...
typedef enum {
    HARD_RESET                          = 0x00,
    ESTABLISH_CONNECTION                = 0x00,
    SOFT_RESET                          = 0x01,
    CHECK_INTERFACE                     = 0x02,
    // ...
    SET_BLINK_TMR1                      = 0x24,

    MESSAGE_TYPE_LIMIT
} MESSAGE_TYPE;
// ...
```

Una vez añadido el tipo de mensaje, sólo queda definir su función en el módulo protocol.{h, c} de forma que configure el temporizador para que el led de IOIO comience a parpadear:

protocol.c

```
// ...
case SET_BLINK_TMR1:
    SetPinDigitalOut(0, 0, 1);
    setTMR1(200);
    enableTMR1();
break;
// ...
```

5.1.2.2.2 Librería

En la librería para Android, el módulo del protocolo es una clase. La clase `IOIOProtocol` contiene todas las variables y métodos necesarios para llevar a cabo la comunicación entre el teléfono y IOIO. Así, haciendo uso de las clases `InputStream`, `OutputStream` y `IncomingHandler`, es posible escribir o leer datos de IOIO.

Al igual que en el módulo del firmware, los mensajes deben estar identificados de la misma forma para ser interpretados correctamente en ambos lados del software. Por tanto, para definir un nuevo mensaje, en primer lugar se debe añadir a los mensajes existentes igual que se hizo en el fichero protocol_defs.h del firmware. Asimismo, también se definirá en esta clase el método que se encargará de mandarle a IOIO la orden de parpadeo del led:

IOIOProtocol.java

```
class IOIOProtocol {
      static final int HARD_RESET                          = 0x00;
      static final int ESTABLISH_CONNECTION                = 0x00;
      static final int SOFT_RESET                          = 0x01;
      static final int CHECK_INTERFACE                     = 0x02;
      // ...
      static final int SET_BLINK_TMR1                      = 0x26;
// ...
      synchronized public void setBlinkTMR1() throws IOException {
          beginBatch();
          writeByte(SET_BLINK_TMR1);
          endBatch();
      }
// ...
```

Para poder añadir la nueva funcionalidad a IOIO es necesario definir en la interfaz `IOIO` el método que proporcionará esta nueva función, ya que esta interfaz es la que proporciona el control sobre todas las funcionalidades de IOIO:

IOIO.java

```
// ...
public void setBlinkTMR1() throws ConnectionLostException;
// ...
```

Finalmente, sólo hay que implementar el método en la clase `IOIOImpl`. Este método, simplemente, llamará a la función definida antes en el protocolo que manda la orden de parpadeo a IOIO:

IOIOImpl.java

```
// ...
@Override
synchronized public void setBlinkTMR1() throws ConnectionLostException {
    try {
        protocol_.setBlinkTMR1();
    } catch (IOException e) {
        throw new ConnectionLostException(e);
    }
}
// ...
```

De esta forma ya es posible usar este nuevo mensaje en nuestra aplicación Android a partir de una instancia de la clase `IOIOImpl`, llamando al método `setBlinkTMR1`.

5.1.2.3 Creación de un mensaje de entrada con parámetros

Para probar el funcionamiento de los parámetros en los mensajes de IOIO se van a crear dos nuevos tipos de mensajes:

- `SET_PIN_BLINK_TMR1`, que pone el pin 0 como salida digital y configura el temporizador Timer1 para el tiempo que le pasemos.
- `SET_BLINK_TMR1`, activa o desactiva el Timer1 en función del parámetro `enable`. Al desactivarlo también apaga el led.

5.1.2.3.1 Firmware

Además de identificar los nuevos mensajes como en la prueba anterior, es necesario definir los argumentos de cada uno en una estructura precisando el número de bytes que ocupa cada argumento. Asimismo, habrá que añadir los mensajes con argumentos a la estructura que define los mensajes entrantes, ya que en este caso los mensajes los envía el dispositivo Android. De esta forma, el fichero protocol_defs.h sería el siguiente:

protocol_defs.h

```
// .
typedef struct PACKED {
    BYTE enable : 1;
    BYTE : 7;
} SET_BLINK_TMR1_ARGS;

typedef struct PACKED {
    WORD miliseg : 12;
    BYTE : 4;
} SET_PIN_BLINK_TMR1_ARGS;

typedef struct PACKED {
    BYTE type;
    union PACKED {
        // ...
        SET_BLINK_TMR1_ARGS                    set_parpadeo_tmr1;
        SET_PIN_BLINK_TMR1_ARGS                set_pin_blink_tmr1;

    } args;
    BYTE __vabuf[72];  // buffer for var args. never access directly!
} INCOMING_MESSAGE;

typedef enum {
    HARD_RESET                          = 0x00,
```

```
    ESTABLISH_CONNECTION                = 0x00,
    SOFT_RESET                          = 0x01,
    // ...
    SET_BLINK_TMR1                      = 0x24,
    SET_PIN_BLINK_TMR1                  = 0x25,

    MESSAGE_TYPE_LIMIT
} MESSAGE_TYPE;
```

Sólo faltaría definir en el protocolo cómo actuará el microcontrolador ante cada mensaje. Cuando se reciba el mensaje `SET_PIN_BLINK_TMR1` se configurará el pin 0 como salida digital, además del temporizador para el tiempo que se le pase como argumento. Con el mensaje `SET_BLINK_TMR1` se activará y desactivará el temporizador. Asimismo, habrá que indicar en el vector de tamaños de mensajes de entrada, el tamaño de los nuevos mensajes con argumentos:

protocol.c

```
// ...
const BYTE incoming_arg_size[MESSAGE_TYPE_LIMIT] = {
  sizeof(HARD_RESET_ARGS),
  sizeof(SOFT_RESET_ARGS),
  // ...
  sizeof(SET_BLINK_TMR1_ARGS),
  sizeof(SET_PIN_BLINK_TMR1_ARGS),
};
// ...
static BOOL MessageDone() {
  switch (rx_msg.type) {
    case HARD_RESET:
      CHECK(rx_msg.args.hard_reset.magic == IOIO_MAGIC);
      HardReset();
      break;
    // ...
    case SET_PIN_BLINK_TMR1:
      SetPinDigitalOut(0,1,1);
      setTMR1(rx_msg.args.set_pin_parpadeo_tmr1.miliseg);
      break;

    case SET_BLINK_TMR1:
      if (rx_msg.args.set_blink_tmr1.enable == 1) {
          enableTMR1();
      }
      if (rx_msg.args.set_blink_tmr1.enable == 0) {
          stopTMR1();
          SetDigitalOutLevel(0, 1);
      }
      break;
// ...
```

5.1.2.3.2 Librería

En la librería de Android hay que añadir al protocolo los nuevos mensajes y los métodos que se encargan de enviarlos con sus parámetros a IOIO. De esta forma, la única diferencia con respecto a la prueba anterior es que ahora se enviará el mensaje y seguidamente los argumentos de este:

IOIOProtocol.java

```
// ...
static final int SET_PIN_BLINK_TMR1              = 0x25;
static final int SET_BLINK_TMR1                  = 0x24;
// ...
synchronized public void setPinBlinkTMR1(int ms) throws IOException {
    beginBatch();
    writeByte(SET_PIN_BLINK_TMR1);
    writeTwoBytes(ms);
    endBatch();
}

synchronized public void setBlinkTMR1(boolean enable) throws IOException {
    beginBatch();
    writeByte(SET_BLINK_TMR1);
    writeByte(enable ? 1 : 0);
    endBatch();
}
// ...
```

Finalmente, sólo queda añadir la nueva funcionalidad en la interfaz `IOIO` definiendo los métodos que proporcionarán estas funciones e implementarlos en la clase IOIOImpl:

IOIOImpl.java

```
// ...
synchronized public void setPinBlinkTMR1(int ms) throws
ConnectionLostException {
    try {
        protocol_.setPinBlinkTMR1(ms);
    } catch (IOException e) {
        throw new ConnectionLostException(e);
    }
}
@Override
synchronized public void setBlinkTMR1(boolean enable) throws
ConnectionLostException {
    try {
        protocol_.setBlinkTMR1(enable);
    } catch (IOException e) {
        throw new ConnectionLostException(e);
    }
}
// ...
```

5.1.2.4 Selección de un pin mediante los parámetros de un mensaje

Hasta ahora, cada vez que se perdía la conexión con IOIO y luego se volvía a conectar, se llamaba a la función `SoftReset` en el FW y se reiniciaba el microcontrolador, por lo que si habíamos programado el led para que parpadeara, este dejaba de parpadear y volvía a su estado inicial. En esta prueba se va a modificar la prueba anterior para poder seleccionar un pin como salida digital. Además, cuando se desconecte y se vuelva a conectar el terminal móvil, IOIO seguirá funcionando como se ha programado.

5.1.2.4.1 Firmware

En el FW habrá que hacer dos cosas: modificar el protocolo para que acepte un nuevo parámetro más y modificar la función `softReset` para que cuando se desconecte el teléfono no se reinicie el pin que hayamos seleccionado.

Partiendo de la prueba anterior, para añadir el parámetro que será la selección del pin solo será necesario añadirlo al mensaje de la siguiente forma en el fichero protocol_defs.h:

protocol_defs.h

```
// ...
typedef struct PACKED {
    WORD miliseg : 12;
    BYTE : 4;
    BYTE pin : 6;
    BYTE : 2;
} SET_PIN_BLINK_TMR1_ARGS;
// ...
```

Una vez definido en el mensaje, se modificará el protocolo para que cuando llegue este mensaje, ponga como salida digital el pin seleccionado:

protocol.c

```
// ...
   case SET_PIN_BLINK_TMR1:
      CHECK(rx_msg.args.set_pin_blink_tmr1.pin < NUM_PINS);
      pin = rx_msg.args.set_pin_blink_tmr1.pin;
      SetPinDigitalOut(rx_msg.args.set_pin_blink_tmr1.pin,1,0);
      setTMR1(rx_msg.args.set_pin_blink_tmr1.miliseg);
      break;
// ...
```

Para hacer que IOIO mantenga el parpadeo una vez se desconecte el terminal, como se ha dicho, será necesario modificar la función `softReset`, que vuelve a IOIO a su estado inicial. La función `softReset` se encuentra en el módulo features.{h, c} y es la siguiente:

features.c

```
void SoftReset(int pin) {
  PRIORITY(7) {
    log_printf("SoftReset()");
    TimersInit();
    PinsInit(pin);
    PWMInit();
    ADCInit();
    UARTInit();
    SPIInit();
    I2CInit();
    InCapInit();
    SequencerInit();
  }
}
```

Para que la salida digital se mantenga hay que modificar la función `PinsInit` que hace que todos los pines de IOIO se pongan como entrada digital. De esta forma, se añadirá un parámetro con el pin seleccionado como salida digital y se le pasará a la función `PinsInit` para que reinicie todos los pins excepto el que se ha seleccionado. Así, la función `PinsInit` modificada será la siguiente:

features.c

```
// ...
static void PinsInit(int pin) {
  int i;
  _CNIE = 0;
  SetPinDigitalOut(0, 1, 1);
  for (i = 1; i < NUM_PINS; ++i) {
    if (i != pin)
        SetPinDigitalIn(i, 0);
  }
  _CNIF = 0;
  _CNIE = 1;
  _CNIP = 1;
}
// ...
```

Sólo queda realizar el cambio en la función principal donde se llama a `SoftReset` para terminar los cambios en el firmware de IOIO:

main.c

```
// ...
void AppCallback(const void* data, UINT32 data_len, int_or_ptr_t arg) {
  if (data) {
    if (!AppProtocolHandleIncoming(data, data_len)) {
      log_printf("Protocol error");
      state = STATE_ERROR;
    }
  } else {
    if (state == STATE_CONNECTED) {
      log_printf("Channel closed");
      SoftReset(pin);
    } else {
      log_printf("Channel failed to open");
    }
    state = STATE_OPEN_CHANNEL;
  }
}
// ...
```

5.1.2.4.2 Librería

Partiendo de la prueba anterior, en el protocolo sólo hay que modificar los métodos con los que se envían los mensajes para añadir el nuevo parámetro de selección de pin.

En la librería de Android hay que añadir al protocolo los nuevos mensajes y los métodos que se encargan de enviarlos con sus parámetros a IOIO. De esta forma, la única diferencia con respecto a la prueba anterior es que ahora se enviará el mensaje y seguidamente los argumentos de este:

IOIOProtocol.java

```
// ...
synchronized public void setPinBlinkTMR1(int pin, int ms) throws IOException
{
    beginBatch();
    writeByte(SET_PIN_BLINK_TMR1);
    writeTwoBytes(ms);
    writeByte(pin);
    endBatch();
}
// ...
```

Finalmente, se añade también el parámetro en la interfaz `IOIO` y en la implementación `IOIOImpl`:

IOIOImpl.java

```
// ...
@Override
synchronized public void setPinParpadeoTMR1(int pin, int ms) throws
ConnectionLostException {
    try {
        protocol_.setPinParpadeoTMR1(pin, ms);
    } catch (IOException e) {
        throw new ConnectionLostException(e);
    }
}
// ...
```

5.1.2.4.3 Aplicación

La aplicación en este caso será algo más compleja y constará de dos actividades para la prueba:

- `MainActivity`: consta de dos entradas de texto, para introducir el tiempo entre interrupciones y el pin que se desea como salida digital, y un botón que redirige a la segunda actividad.

Figura 5-1. Actividad `MainActivity` para la prueba de mensajes con argumento

- `ParpadeoLedActivity`: mediante un `ToggleButton`, es decir, un botón con dos estados, modifica el parámetro `enable` y activa y desactiva el parpadeo del led para los parámetros establecidos en la actividad `MainActivity`.

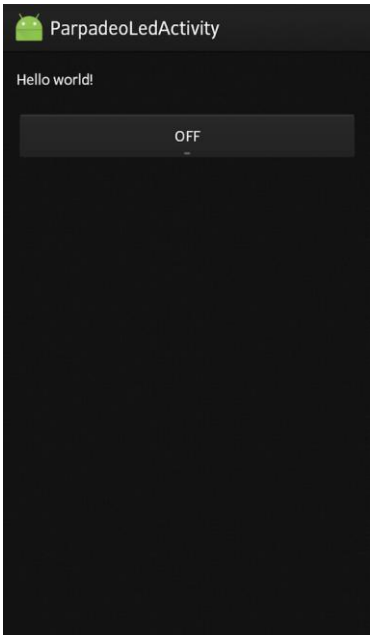


Figura 5-2. Actividad `ParpadeoLedActivity` para la prueba de mensajes con argumento

5.1.2.5 Creación de un mensaje de salida desde IOIO

Hasta este momento sólo se ha trabajado con mensajes de entrada en IOIO, es decir, mensajes enviados desde el terminal Android hacia el microcontrolador para que este realice una determinada acción. Si se desea recibir un mensaje de vuelta con un determinado dato del microcontrolador (por ejemplo, el contenido de un registro, el estado de una variable global, etc.), es necesario definir un mensaje de salida en IOIO y gestionar la recepción de ese mensaje en la librería de Android.

Para comprobar el funcionamiento de los mensajes de salida en IOIO, se va a realizar una prueba en la que se enviará un valor constante y se comprobará que en el dispositivo Android se recibe ese mismo valor y la transmisión es correcta.

5.1.2.5.1 Firmware

Se van a defirnir dos nuevos mensajes en el protocolo de IOIO:

- `SET_RETURN_CONSTANT_STATUS`: este será un mensaje de entrada que enviará el terminal Android para dar la orden al microcontrolador de que se le devuelva el valor constante que se quiere enviar.
- `REPORT_CONSTANT_STATUS`: mensaje de salida que constará de un argumento que llevará el valor constante que se quiere transmitir al smartphone.

Por tanto, estos cambios se reflejarán en el fichero protocol_defs.h de la siguiente forma:

protocol_defs.h

```
// ...
typedef struct PACKED {
    BYTE constant;
} REPORT_CONSTANT_STATUS _ARGS;
// ...
typedef struct PACKED {
  BYTE type;
  union PACKED {
    ESTABLISH_CONNECTION_ARGS               establish_connection;
    REPORT_DIGITAL_IN_STATUS_ARGS           report_digital_in_status;
    // ...
    REPORT_CONSTANT_STATUS_ARGS             report_constant_status;
  } args;
} OUTGOING_MESSAGE;

typedef enum {
    HARD_RESET                          = 0x00,
    ESTABLISH_CONNECTION                = 0x00,
    SOFT_RESET                          = 0x01,
    // ...
    SET_RETURN_CONSTANT                 = 0x24,
    REPORT_CONSTANT_STATUS              = 0x25,

    MESSAGE_TYPE_LIMIT
} MESSAGE_TYPE;
// ...
```

En el módulo del protocolo será necesario indicar que cuando se reciba el mensaje `SET_RETURN_CONSTANT`, se debe realizar el envío del mensaje `REPORT_CONSTANT_STATUS`, para lo que se creará una función `ReportConstatStatus()` que realizará el envío del mensaje. Asimismo, hay que indexar el mensaje de salida creado en la tabla `outgoing_arg_size` con el tamaño del argumento del mensaje.

protocol.c

```
const BYTE outgoing_arg_size[MESSAGE_TYPE_LIMIT] = {
  sizeof(ESTABLISH_CONNECTION_ARGS),
  sizeof(SOFT_RESET_ARGS),
  sizeof(CHECK_INTERFACE_RESPONSE_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(REPORT_DIGITAL_IN_STATUS_ARGS),
  // …
  sizeof(REPORT_CONSTANT_STATUS_ARGS)
};
// ...
static BOOL MessageDone() {
  switch (rx_msg.type) {
    // ...
    case SET_RETURN_ANALOG_OFFLINE_STATUS:
      ReportConstantStatus();
    break;
//...
}
//...
void ReportConstantStatus() {
```

```
  OUTGOING_MESSAGE msg;
  msg.type=REPORT_CONSTANT_STATUS;
  msg.args.report_constant_status.constant=1;
  AppProtocolSendMessage(&msg);
}
```

5.1.2.5.2 Librería

En la librería, en esta ocasión, habrá que realizar cambios diferentes a los de las pruebas anteriores, ya que será necesario gestionar un mensaje de entrada al terminal. Para ello, en primer lugar se definirá el método que manejará la variable de entrada en la interfaz `IncomingHandler`, que se encuentra dentro de la clase `IOIOProtocol`:

IOIOProtocol.java

```
//...
	public interface IncomingHandler {
		public void handleEstablishConnection(byte[] hardwareId, byte[]
bootloaderId, byte[] firmwareId);

		public void handleConnectionLost();
		//...
		public void handleConstant(int constant);
	}
//...
```

La implementación del método se lleva a cabo en la clase `IncomingState`, y simplemente copia el valor de la variable que obtiene del método en una variable de instancia que posteriormente será la que se obtenga a través de la aplicación.

IncomingState.java

```
class IncomingState implements IncomingHandler {
	//...
	public int constant_;
	//...
	@Override
	public void handleConstant(int constant) {
		constant_=constant;
	}
}
```

Una vez que este método se ha implementado, hay que realizar la lectura del mensaje de entrada. Los mensajes de entrada son gestionados por el hilo de recepción de mensajes `IncomingThread`, en la clase `IOIOProtocol`.

```
//...
	class IncomingThread extends Thread {
		//...
		@Override
		public void run() {
			super.run();
			setPriority(MAX_PRIORITY);
			int arg1;
			int arg2;
			//...
			try {
				while (true) {
					switch (arg1 = readByte()) {
					//...
					case REPORT_ANALOG_OFFLINE_STATUS:
```

```
                                    arg1=readByte();
                                    handler_.handleAnalogOffline(arg1);
                                break;
//...
```

Finalmente, quedaría definir en la interfaz `IOIO` el método con el que se obtendrá el valor de la constante en la aplicación e implementarlo en `IOIOImpl`. Esto se obtendrá mediante el objeto `incomingState_` y la variable que se definió anteriormente.

IOIOImpl.java

```
//...
      @Override
      public int getConstant() throws ConnectionLostException {
            try{
                  return incomingState_.constant_;
            } finally {

            }
      }
//...
```

5.1.3 Convertidor Analógico Digital

Para probar el desarrollo del convertidor A/D, la primera prueba que se realizó, antes de obtener una serie de muestras, consistió en realizar una lectura simple. De tal forma, el principal valor de esta prueba consistía en comprobar la correcta configuración de los registros del convertidor y ver que se obtenía un dato preciso de lo que se estaba leyendo.

5.1.3.1 Firmware

La configuración de los registros del convertidor, así como el temporizador, mensajes y demás es prácticamente idéntica a lo que ya se ha visto en el capítulo 3 de este documento. Por tanto, este apartado se centrará en las diferencias respecto a esa parte. Básicamente, en este caso en lugar de enviar una ristra de muestras se enviará una sola, por lo que el formato del mensaje cambiará y en el argumento sólo se enviarán dos bytes, ya que el registro con el resultado de la conversión es de 10 bits.

protocol_defs.h

```
//...
typedef struct PACKED {
    WORD ADCResult;
} REPORT_ANALOG_OFFLINE_STATUS_ARGS;
//...
```

5.1.3.2 Librería

En la librería, tomando como referencia la prueba realizada anteriormente, donde se recibía una constante, en este caso habrá que tener en cuenta que se recibirán dos bytes y que habrá que reorganizarlos para obtener el valor del registro.

IOIOProtocol.java

```
//...
                                case REPORT_ANALOG_OFFLINE_STATUS:
                                      int ADCResult;
                                      arg1=readByte();
                                      arg2=readByte();
                                      ADCResult=arg1 | (arg2<<8);
                                      handler_.handleAnalogOffline(ADCResult);
                                 break;
//...
```

Por lo demás, se procede de la misma forma que en la prueba anterior.

5.1.3.3 Aplicación

La aplicación mostrará en pantalla la resultado de la conversión convertido a tensión. Para la prueba, se ha utilizado un potenciómetro con una variación de tensión de entre 0 y 3,3 V. De esta forma, hay que cerciorar que la tensión que se tiene a la entrada del pin, es convertida de forma correcta y corresponde con el dato que se muestre en pantalla.

Esta consta de una actividad que hereda de IOIOActivity para implementar sus objetos y variables. El procedimiento será básicamente el mismo que el llevado a cabo en el capítulo 3. En este caso, se tendrán dos botones, uno que mandará la orden de leer y otro que dará la orden de recoger los datos. Así, como la aplicación no se basará en estados, si no en acciones, se necesitará una copia del objeto ioio_ fuera de la clase Looper. El código de la aplicación sería el siguiente:

MainActivity.java

```
//...
    public void readADCOffline(View view) throws ConnectionLostException,
InterruptedException {
        try {
            myioio.setAnalogOffline();
            myioio.disconnect();
        } catch (ConnectionLostException e) {
            throw e;
        }
    }

    public void returnADCResult(View view) throws ConnectionLostException,
InterruptedException {
        try {
            myioio.setReturnAnalogOfflineStatus();
            Thread.sleep(200);
            final int reading = myioio.getADCResult();
            Thread.sleep(200);
            final float reading01=reading/1023.f;
            final float voltage=reading01*3.3f;
            setText(Float.toString((voltage))+ " V");
            myioio.disconnect();
        } catch (ConnectionLostException e) {
            throw e;
        }

    class Looper extends BaseIOIOLooper {

        @Override
        protected void setup() throws ConnectionLostException {
            myioio = ioio_;
```

```
        }

        @Override
        public void loop() throws ConnectionLostException {

        }

        @Override
        public void disconnected() {

        }
    }
//...
```

Se comprobará el correcto funcionamiento del conversor obteniendo diversos valores. Cuando el potenciómetro se encuentre en un extremo, por pantalla deberá mostrar una tensión de 3,3 V. Cuando se encuentre en el otro extremo 0 V. En cualquier otro caso se mostrarán valores intermedios entre 0 y 3,3 V. De esta forma, se comprueba que el desarrollo del convertidor es correcto.

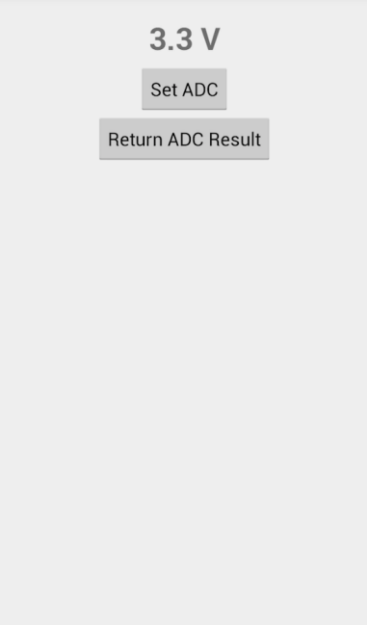


Figura 5-3. Aplicación cuando se realiza una lectura simple de 3,3 V

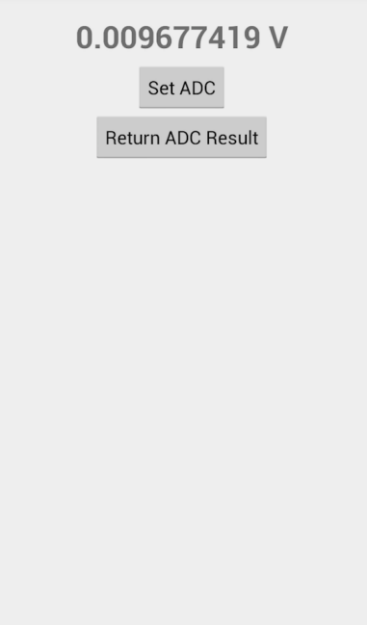


Figura 5-4. Aplicación cuando se realiza una lectura simple de 0 V

5.2 Pruebas finales

Hasta ahora se han realizado pruebas de tres tipos distintos:

- Pruebas de unidad, en las que se ha ido comprobando cada módulo (temporizador, conversor A/D, tipos de mensajes, etc. de forma individual para verificar que el diseño y su programación son correctos.
- De integración, donde componentes que se han visto anteriormente deben ensamblarse para formar el software completo. De este tipo también se ha realizado alguna prueba cuando se ha probado el convertidor A/D con una muestra simple, donde se han integrado temporizador, convertidor, mensajes de entrada y salida, etc.
- De sistema, donde el software se prueba junto con el hardware (IOIO y terminal Android) y se verifica el funcionamiento y rendimiento global. También probado en apartados anteriores.

En este apartado, además de pruebas de integración y de sistema, las principales pruebas serán de validación, enfocadas en validar el cumplimiento de requisitos y comprobar que se construyó lo que se quería. Básicamente, estas pruebas se enfocan en las acciones y salidas visibles para el usuario. Por lo cual, teniendo en cuenta lo desarrollado en el capítulo tres, se realizarán tres pruebas:

- Prueba 1: se realizarán menos de 125 conversiones, de forma que sólo se recibe un mensaje.
- Prueba 2: se realizarán más de 125 y menos de 1125 conversiones, de forma que no se llenará la memoria RAM y se recibirá más de un mensaje.
- Prueba 3: se completará la memoria con las muestras.

En cada prueba habrá que comprobar la usabilidad de la aplicación, es decir, que se pueden contemplar los resultados en la gráfica, que los datos correctos y que estos se almacenan en el fichero CSV cuando el usuario desea.

5.2.1 Prueba 1

Durante esta prueba el potenciómetro irá variando la tensión de entre 0 y 3,3 V durante menos de 125 segundos, de forma que el terminal sólo recibirá un mensaje que no estará completo. Tras programar la conversión pulsando el botón Set ADC y posteriormente pararla con Stop ADC se observa la siguiente gráfica con la evolución de la señal de tensión de entrada en el pin 36 respecto al tiempo.

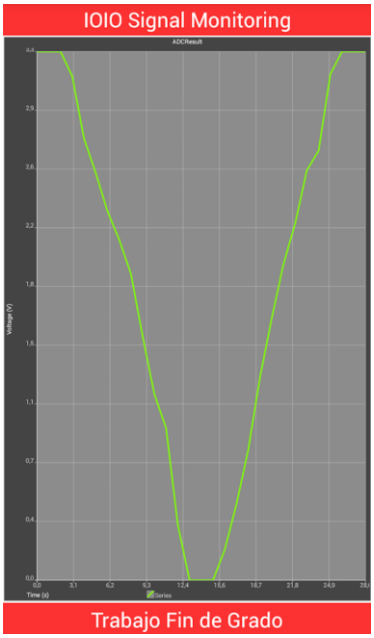


Figura 5-5. Evolución de la señal en Prueba1

La gráfica corresponde con la evolución de la señal de entrada proporcionada por el ponteciómetro. Por tanto, se puede verificar la correcta programación. Queda por ver si el resultado almacenado en el fichero CSV corresponde con las muestras y se verifica el funcionamiento. La siguiente ristra de valores corresponden a los almacenados en el registro del convertidor, sin convertirlos a valores de tensión:

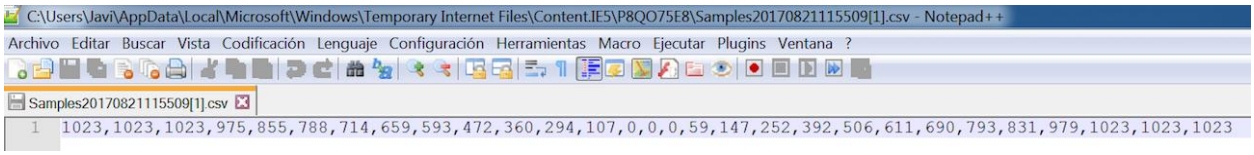


Figura 5-6. Fichero CSV generado en Prueba1

Se puede observar que se obtienen 28 muestras. Si el valor de 1023 corresponde a 3,3 V y el de 0 a 0 V, se puede verificar que el fichero se ha generado con los datos correctos.

5.2.2 Prueba 2

En esta prueba, se ha realizado un escaneo de la señal durante 335 segundos. De esta forma, se han enviado tres paquetes de datos y se ha obtenido la siguiente gráfica:

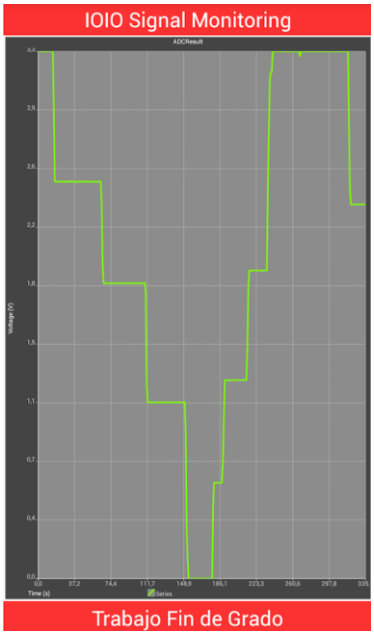


Figura 5-7. Evolución de la señal en Prueba2

El fichero CSV contiene las 335 muestras sin formato y es el siguiente:

Samples20170821121500.csv

```
1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,10
23,903,771,770,771,770,770,770,770,771,770,770,770,770,771,770,770,771,770,77
1,770,770,770,770,770,770,770,771,770,770,770,770,771,770,770,771,770,770,770
,770,770,770,770,770,770,770,770,770,771,770,724,604,573,572,573,573,573,573,
573,573,573,573,573,573,573,573,573,573,573,573,573,573,573,573,573,573,574,5
73,573,573,573,573,573,573,573,573,573,574,573,573,573,573,573,573,574,554,37
8,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342
,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342,342,
342,281,95,35,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,140,186,186,1
86,186,186,186,186,186,186,186,225,314,385,385,385,385,384,385,385,385,385,385,38
5,385,385,385,385,385,385,385,385,385,385,385,385,443,560,598,598,597,598,598
,598,598,598,598,598,598,598,597,597,598,598,598,598,598,755,866,966,983,983,
1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,10
23,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1013,1023,1023
,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1
023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,102
3,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,
1023,889,758,726,726,726,726,726,726,726,726,726,726,726,726,726,726,726,726
```

Ambos datos se corresponden a la variación de la resistencia realizada en el potenciómetro, por lo que se puede validar la aplicación cuando se envía más de un mensaje.

5.2.3 Prueba 3

La última prueba consiste en llenar todo el vector de muestras, para lo cual se mantendrá IOIO muestreando durante 20 minutos a una muestra por segundo (1250 muestras). Así, desde IOIO se enviarán 10 paquetes, cada uno con 125 muestras. La evolución de la tensión respecto al tiempo en la gráfica de la aplicación es la siguiente:

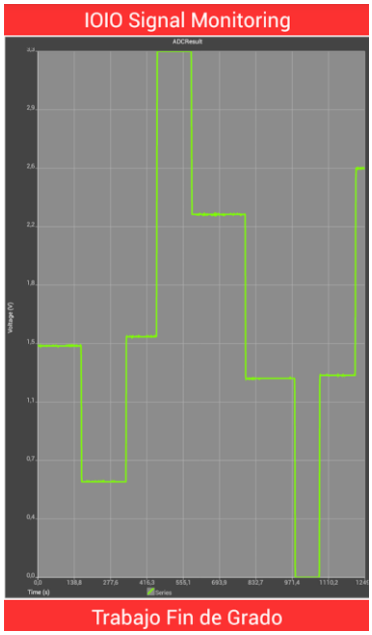


Figura 5-8. Evolución de la señal en Prueba3

Se comprueba que, efectivamente, en el fichero CSV se han guardado las 1250 muestras y que los valores se corresponden a los de la gráfica y la señal de entrada.

5.3 Herramientas de depuración

Durante cada una de las pruebas llevadas a cabo han surgido errores y ha sido necesario identificarlos y llevar a cabo diversos cambios para corregirlos. En función de dónde fuera necesario depurar, se han utilizado diversas herramientas que se comentarán en este apartado.

5.3.1 Depuración en IOIO

Para depurar errores en el microcontrolador, la única herramienta que se tenía a disposición era el led de IOIO (led STAT). De tal forma, si se necesitaba conocer cuándo se accedía a una rutina de interrupción o a cualquier otra función, la forma de averiguarlo era encendiendo y apagando el led del microcontrolador.

Así, si por ejemplo se desea averiguar si el microcontrolador está ejecutando la rutina de interrupción del convertidor A/D, se procede de la siguiente forma:

1) En primer lugar, cuando se recibe el mensaje que activa la conversión, se programa el pin de estado como salida digital, permaneciendo apagado:

adc_offline.c

```
//...
void ADCOfflineSetScan() {
    SetPinDigitalOut(0, 1, 1);
    ADCOfflineStop();

    AD1CON1 = 0x0000;
    AD1CON2 = 0x0000;
    AD1CON3 = 0x1F01;
    AD1CHS  = 0x0000;
    AD1CSSL = 0x0001;

    ADCOfflineStart();
}
//...
```

2) El siguiente paso sería encender y apagar el led en la rutina de interrupción, para lo cual, el firmware del microcontrolador proporciona las funciones `led_on()` y `led_off()`.

adc.c

```
//...
void __attribute__((__interrupt__, auto_psv)) _ADC1Interrupt() {
  led_on();

  if (ADCtype==0)
    ScanDoneInterruptTrigger();
  else {
    if(samplingIndex<NUM_SAMPLES*NUM_MSG) {
        ADCResult[samplingIndex]=ADC1BUF0;
        samplingIndex++;
    }
    else if(samplingIndex==NUM_SAMPLES*NUM_MSG) {
        disableTMR1();
        ADCOfflineStop();
    }
  }
  _ADON = 0;
  _AD1IF = 0;

  int k=0;
  for(k=0; k<500;k++){}
    led_off();
}
//...
```

De esta forma, es posible conocer cuándo cambia de valor una variable, cuándo entra en una función, etc. sin más ayuda que un led y el código del firmware.

5.3.2 Depuración en la librería y la aplicación

Para depurar en Android y en la librería de IOIO se ha utilizado la API `Log` de Android [21]. Esta API permite enviar registros de salida, de forma que se pueda observar el valor de las variables en un punto dado, cuándo se accede a un método, etc.

La forma de proceder, tanto en la librería como en la aplicación es la siguiente:

1) Se declara una variable de clase etiqueta (`TAG`) que identifique la clase, módulo, librería, …, de la que proceden los registros.

```
public static final String TAG = "IOIOAPP";
```

2) A partir de este momento, utilizando los métodos de registro, se puede imprimir como salida cualquier variable o texto. Por ejemplo, en el método `returnADCResult` de la aplicación, se utiliza así:

```
//...
    public void returnADCResult(View view) throws ConnectionLostException,
InterruptedException {
        int[] msgResult;
        int numMsg = 0;
        while (numMsg != 1) {
            myioio.setReturnAnalogOfflineStatus();
            Thread.sleep(300);
            numMsg = myioio.getNumMsg();
            Thread.sleep(300);
            Log.v(TAG, "numMsg= " + numMsg);
            msgResult = myioio.getADCResult();
            Thread.sleep(300);
            Log.v(TAG, "ADCResult" + numMsg + "= " +
Arrays.toString(msgResult));
            msgResultList.add(myioio.getADCResult());
        }
        //...
```

3) Por último, para ver los mensajes de registro, se necesita una aplicación que los muestre. Para este proyecto se ha utilizado la aplicación *CatLog* [22]. Esta aplicación permite localizar los registros a partir de su etiqueta, lo que facilita enormemente la depuración de la aplicación y la librería.

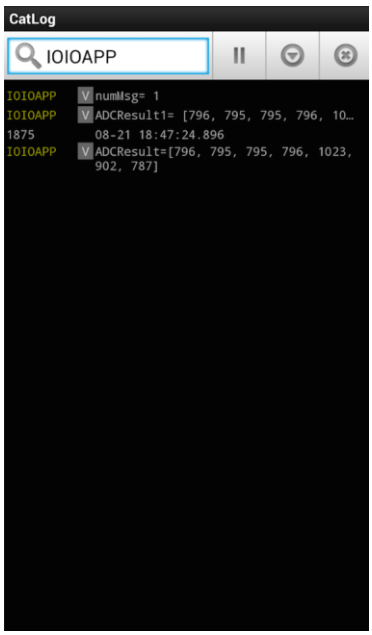


Figura 5-9. Registros de la aplicación en *CatLog*

6 CONCLUSIONES Y DESARROLLOS FUTUROS

Sólo cerrando las puertas detrás de uno se abren
ventanas hacia el porvenir.

- Françoise Sagan -

En este capítulo se va a realizar una valoración del sistema de monitorización de señales, señalando los problemas encontrados durante su desarrollo; las conclusiones que se pueden sacar, tanto de la herramienta en sí como personales; y los desarrollos futuros que se podrán seguir para mejorar la aplicación y el protocolo, de forma que se tenga un sistema más versátil.

6.1 Problemas encontrados

Durante el desarrollo del proyecto se han encontrado las siguientes dificultades:

- Escasez de información. A diferencia de otras plataformas como Arduino o Raspberry Pi, IOIO sólo cuenta con la información de su desarrollador en github y un foro, por lo que ha sido más complicado adentrarse en el desarrollo del firmware y la librería.
- Configuración de las plataformas de desarrollo. Con la falta de información, fue complicado configurar cada una de estas. Por ejemplo, en MPLAB fue necesario buscar un compilador compatible con el proyecto distinto al cual este utilizaba originariamente.
- Estudio del protocolo. Al tener ya un protocolo de comunicaciones establecido, fue necesario estudiarlo a fondo y entenderlo para poder realizar modificaciones sobre este. Al ser bastante complejo, tanto en el lado del microcontrolador, como en el del firmware, surgieron complicaciones durante las primeras pruebas.
- Almacenamiento de datos en memoria RAM. Aunque no es lo ideal, IOIO posee una memoria flash con poca capacidad (del orden de 100K) y además tiene un ciclo de escritura-borrado limitado, por lo que guardar los datos de forma persistente no era una buena opción a considerar. Al guardarlos en memoria RAM, aunque no se tiene esa persistencia, es mucho más sencillo el manejo de datos y no se está limitada en cuanto al acceso.
- Aplicación basada en estados. Las aplicaciones de IOIO se basan en estados, de forma que, hasta antes de modificar el software, el terminal debía estar conectado a IOIO todo el tiempo para que este funcionase. Cambiar de aplicación a una basada en acciones, por tanto, no fue sencillo y requirió de bastante estudio en ambos lados de la comunicación.

6.2 Conclusiones

Tras el fin de este proyecto, se pueden extraer las siguientes conclusiones en sentido técnico:

- La elección de IOIO y Android como plataformas de desarrollo proporcionan un entorno libre de desarrollo a través del cual se tiene cierta flexibilidad para llevar a cabo tu propia solución técnica.
- La solución presentada en este proyecto aporta algo más de versatilidad a la dada por defecto por IOIO, en la cual era necesaria tener el terminal Android y el microcontrolador siempre comunicados para llevar a cabo cualquier tipo de función.
- En general, el proyecto desarrollado ofrece flexibilidad de forma que sólo son necesarios un terminal Android y una batería para proporcionar alimentación a IOIO y a un determinado sensor.

Como conclusiones personales, se puede decir que este proyecto ha proporcionado la oportunidad de aplicar y profundizar los conocimientos adquiridos durante todo el Grado de Ingeniería de las Tecnologías de la Telecomunicación, desde la programación en sistemas electrónicos digitales y la programación en Android, pasando por la Ingeniería de Software, hasta la aplicación de conocimientos en administración de Sistemas Operativos Linux.

6.3 Desarrollos futuros

El sistema de monitorización de señales cumple con los objetivos marcados. Sin embargo, su potencial es relativamente bajo debido a la poca capacidad de memoria disponible y a la no persistencia de los datos, lo que hace que haya que plantear las siguientes líneas futuras para el proyecto:

- Añadir persistencia y mayor almacenamiento de datos al sistema. Para lo cual habría que conectar algún tipo de memoria persistente externa y desarrollar un controlador o *driver* para su uso con IOIO.
- Incorporar conversión simultánea en varios canales con elección del pin de entrada analógica en IOIO, de forma que se puedan monitorizar varios sensores a la vez.
- Mejorar el protocolo para que la aplicación Android no se tenga que basar por defecto en estados y sea más eficiente en este sentido.
- Integrar los cambios en la interfaz bluetooth para utilizar las nuevas funcionalidades sin conexión por cable microUSB.

REFERENCIAS

[1] A. L. Sucasas, «España, enganchada al ‘smartphone’,» *El País*, 2015.

[2] J. M. Fornes Rumbao, Diseño de Aplicaciones Móviles, Sevilla: Universidad de Sevilla, 2015.

[3] J. Yarow, «Business Insider,» 2014. [En línea]. Available: http://www.businessinsider.com/androids-share-of-the-computing-market-2014-3. [Último acceso: 2016].

[4] Android Developers, «Android Lollipop,» 2015. [En línea]. Available: http://developer.android.com/about/versions/lollipop.html. [Último acceso: 2016].

[5] Cinco Días, 2016. [En línea]. Available: http://cincodias.com/cincodias/2016/06/15/tecnologia/1465987235_750846.html. [Último acceso: 2016].

[6] Microsoft, 2016. [En línea]. Available: https://msdn.microsoft.com/windows/uwp/get-started/whats-a-uwp. [Último acceso: 2016].

[7] Y. Ben-Tsvi, «Github: IOIO Wiki,» 2012. [En línea]. Available: https://github.com/ytai/ioio/wiki. [Último acceso: 2016].

[8] S. Monk, Making Android Accessories with IOIO, O'REILLY, 2012.

[9] seedstudio.com, «Seeeduino ADK Main Board,» [En línea]. Available: https://www.seeedstudio.com/Seeeduino-ADK-Main-Board-p-846.html#. [Último acceso: 2016].

[10] Arduino, [En línea]. Available: https://www.arduino.cc/en/Main/arduinoBoardUno. [Último acceso: 2016].

[11] M. Perez Esteso, «Geekytheory,» 2013. [En línea]. Available: https://geekytheory.com/android-arduino-andruino. [Último acceso: 2016].

[12] Appcelerator, [En línea]. Available: http://www.appcelerator.com. [Último acceso: 2016].

[13] BYOD, «BYOD (Build Your Own Drone),» [En línea]. Available: http://www.buildyourowndrone.co.uk/phonedrone-board-for-android.html. [Último acceso: 2016].

[14] Microchip Technology, [En línea]. Available: http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=DM240415. [Último acceso: 2016].

[15] Microchip Technology, [En línea]. Available: http://www.microchip.com/mplab/mplab-x-ide. [Último acceso: 2016].

[16] Wikipedia, «Eclipse,» 2016. [En línea]. Available: https://en.wikipedia.org/wiki/Eclipse_(software).

[Último acceso: 2016].

[17] Android Developers, «Android Studio,» 2016. [En línea]. Available: https://developer.android.com/studio/intro/index.html. [Último acceso: 2016].

[18] «Androidplot,» 2017. [En línea]. Available: http://androidplot.com.

[19] R. S. Pressman, Ingeniería del software. Un enfoque práctico., McGraw Hill, 2010.

[20] I. Roman Martinez, Ingeniería del Software, Sevilla: Universidad de Sevilla, 2014.

[21] «Android Developers,» 2017. [En línea]. Available: https://developer.android.com/reference/android/util/Log.html.

[22] N. Lawson, «Github,» 2017. [En línea]. Available: https://github.com/nolanlawson/Catlog.

[23] «PIC24FJ256GB210 Family Data Sheet,» 2010. [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/39975a.pdf. [Último acceso: 2017].

[24] Y. Ben-tsvi, «IOIO OTG schematic,» 2012. [En línea]. Available: https://cdn.sparkfun.com/datasheets/Dev/Android/IOIO-OTG-v20b.pdf. [Último acceso: 2017].

[25] «PIC24F Family Reference Manual. Section 14. Timers,» 2010. [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/39704a.pdf. [Último acceso: 2017].

[26] «PIC24F Family Reference Manual. Section 17. 10-Bit A/D Converter,» 2010. [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/39705b.pdf. [Último acceso: 2017].

ANEXO A: DATASHEETS Y ESQUEMÁTICOS

En este apéndice se muestran los datasheets y esquemáticos utilizados para el desarrollo de este proyecto, con las características generales. Para su consulta completa y detallada se deben acudir a las referencias.

1. Datasheet de la familia PIC24FJ256GB210 de microcontroladores

Fuente: [23]

MICROCHIP **PIC24FJ256GB210 FAMILY**

64/100-Pin, 16-Bit Flash Microcontrollers with USB On-The-Go (OTG)

Universal Serial Bus Features:

- USB v2.0 On-The-Go (OTG) Compliant
- Dual Role Capable – Can act as either Host or Peripheral
- Low-Speed (1.5 Mbps) and Full-Speed (12 Mbps) USB Operation in Host mode
- Full-Speed USB Operation in Device mode
- High-Precision PLL for USB
- Supports up to 32 Endpoints (16 bidirectional):
  - USB module can use the internal RAM location from 0x800 to 0xFFFF as USB endpoint buffers
- On-Chip USB Transceiver with Interface for Off-Chip Transceiver
- Supports Control, Interrupt, Isochronous and Bulk Transfers
- On-Chip Pull-up and Pull-Down Resistors

Peripheral Features:

- Enhanced Parallel Master Port/Parallel Slave Port (EPMP/PSP):
  - Direct access from CPU with an Extended Data Space (EDS) interface
  - 4, 8 and 16-bit wide data bus
  - Up to 23 programmable address lines
  - Up to 2 chip select lines
  - Up to 2 Acknowledgement lines (one per chip select)
  - Programmable address/data multiplexing
  - Programmable address and data Wait states
  - Programmable polarity on control signals

Peripheral Features (Continued):

- Peripheral Pin Select:
  - Up to 44 available pins (100-pin devices)
- Three 3-Wire/4-Wire SPI modules (supports 4 Frame modes)
- Three I^2C™ modules Supporting Multi-Master/Slave modes and 7-Bit/10-Bit Addressing
- Four UART modules:
  - Supports RS-485, RS-232, LIN/J2602 protocols and IrDA®
- Five 16-Bit Timers/Counters with Programmable Prescaler
- Nine 16-Bit Capture Inputs, each with a Dedicated Time Base
- Nine 16-Bit Compare/PWM Outputs, each with a Dedicated Time Base
- Hardware Real-Time Clock and Calendar (RTCC)
- Enhanced Programmable Cyclic Redundancy Check (CRC) Generator
- Up to 5 External Interrupt Sources

PIC24FJ Device	Pins	Program Memory (bytes)	SRAM (bytes)	Remappable Peripherals					I^2C™	10-Bit A/D (ch)	Comparators	CTMU	EPMP/PSP	RTCC	USB OTG
				Remappable Pins	16-Bit Timers	IC/OC PWM	UART w/IrDA®	SPI							
PIC24FJ128GB206	64	128K	96K	29	5	9/9	4	3	3	16	3	Y	Y	Y	Y
PIC24FJ256GB206	64	256K	96K	29	5	9/9	4	3	3	16	3	Y	Y	Y	Y
PIC24FJ128GB210	100/121	128K	96K	44	5	9/9	4	3	3	24	3	Y	Y	Y	Y
PIC24FJ256GB210	100/121	256K	96K	44	5	9/9	4	3	3	24	3	Y	Y	Y	Y

© 2010 Microchip Technology Inc.

PIC24FJ256GB210 FAMILY

High-Performance CPU

- Modified Harvard Architecture
- Up to 16 MIPS Operation at 32 MHz
- 8 MHz Internal Oscillator
- 17-Bit x 17-Bit Single-Cycle Hardware Multiplier
- 32-Bit by 16-Bit Hardware Divider
- 16 x 16-Bit Working Register Array
- C Compiler Optimized Instruction Set Architecture with Flexible Addressing modes
- Linear Program Memory Addressing, up to 12 Mbytes
- Data Memory Addressing, up to 16 Mbytes:
  - 2K SFR space
  - 30K linear data memory
  - 66K extended data memory
  - Remaining (from 16 Mbytes) memory (external) can be accessed using extended data Memory (EDS) and EPMP (EDS is divided into 32-Kbyte pages)
- Two Address Generation Units for Separate Read and Write Addressing of Data Memory

Power Management:

- On-Chip Voltage Regulator of 1.8V
- Switch between Clock Sources in Real Time
- Idle, Sleep and Doze modes with Fast Wake-up and Two-Speed Start-up
- Run Mode: 800 µA/MIPS, 3.3V Typical
- Sleep mode Current Down to 20 µA, 3.3V Typical
- Standby Current with 32 kHz Oscillator: 22 µA, 3.3V Typical

Analog Features:

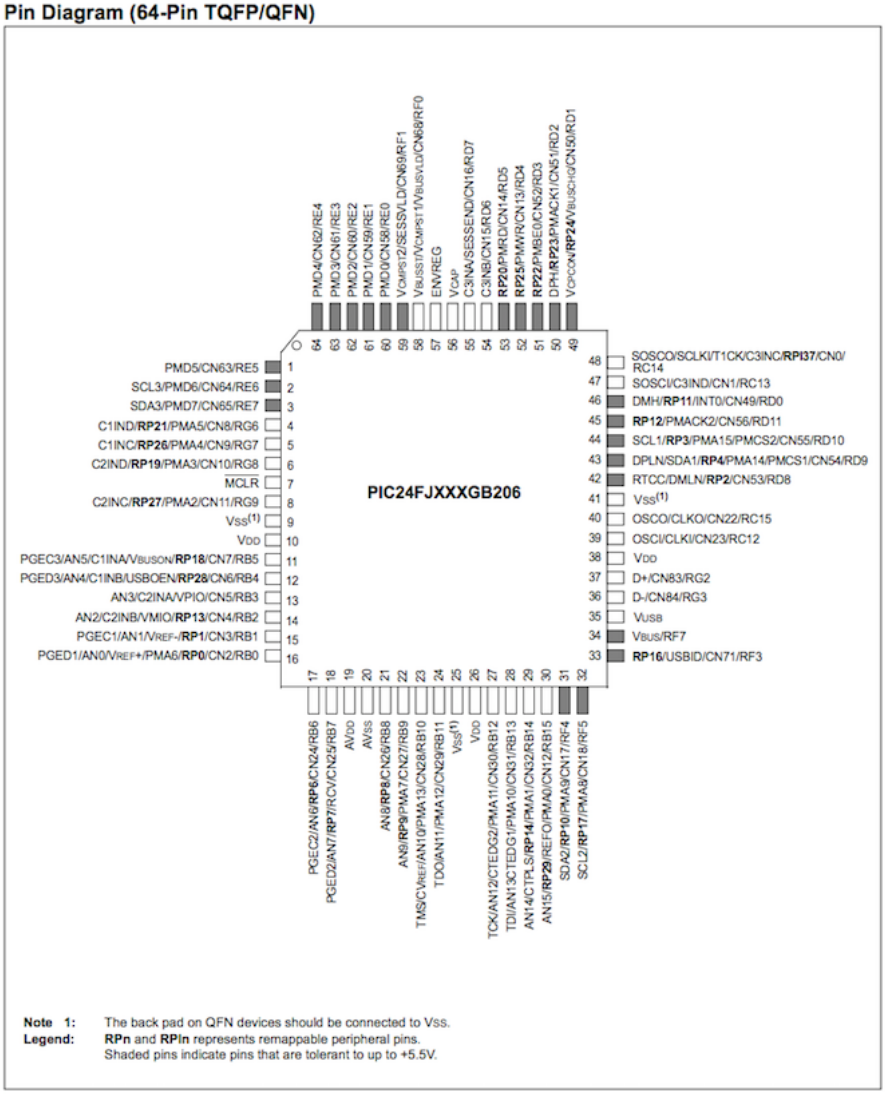
- 10-Bit, up to 24-Channel Analog-to-Digital (A/D) Converter at 500 ksps:
  - Operation is possible in Sleep mode
  - Band gap reference input feature
- Three Analog Comparators with Programmable Input/Output Configuration
- Charge Time Measurement Unit (CTMU):
  - Supports capacitive touch sensing for touch screens and capacitive switches
  - Minimum time measurement setting at 100 ps
- Available LVD Interrupt V_{LVD} Level

Special Microcontroller Features:

- Operating Voltage Range of 2.2V to 3.6V
- 5.5V Tolerant Input (digital pins only)
- Configurable Open-Drain Outputs on Digital I/O Ports
- High-Current Sink/Source (18 mA/18 mA) on all I/O Ports
- Selectable Power Management modes:
  - Sleep, Idle and Doze modes with fast wake-up
- Fail-Safe Clock Monitor (FSCM) Operation:
  - Detects clock failure and switches to on-chip, FRC oscillator
- On-Chip LDO Regulator
- Power-on Reset (POR) and Oscillator Start-up Timer (OST)
- Brown-out Reset (BOR)
- Flexible Watchdog Timer (WDT) with On-Chip Low-Power RC Oscillator for Reliable Operation
- In-Circuit Serial Programming™ (ICSP™) and In-Circuit Debug (ICD) via 2 Pins
- JTAG Boundary Scan Support
- Flash Program Memory:
  - 10,000 erase/write cycle endurance (minimum)
  - 20-year data retention minimum
  - Selectable write protection boundary
  - Self-reprogrammable under software control
  - Write protection option for Configuration Words

© 2010 Microchip Technology Inc.

Pin Diagram (64-Pin TQFP/QFN)

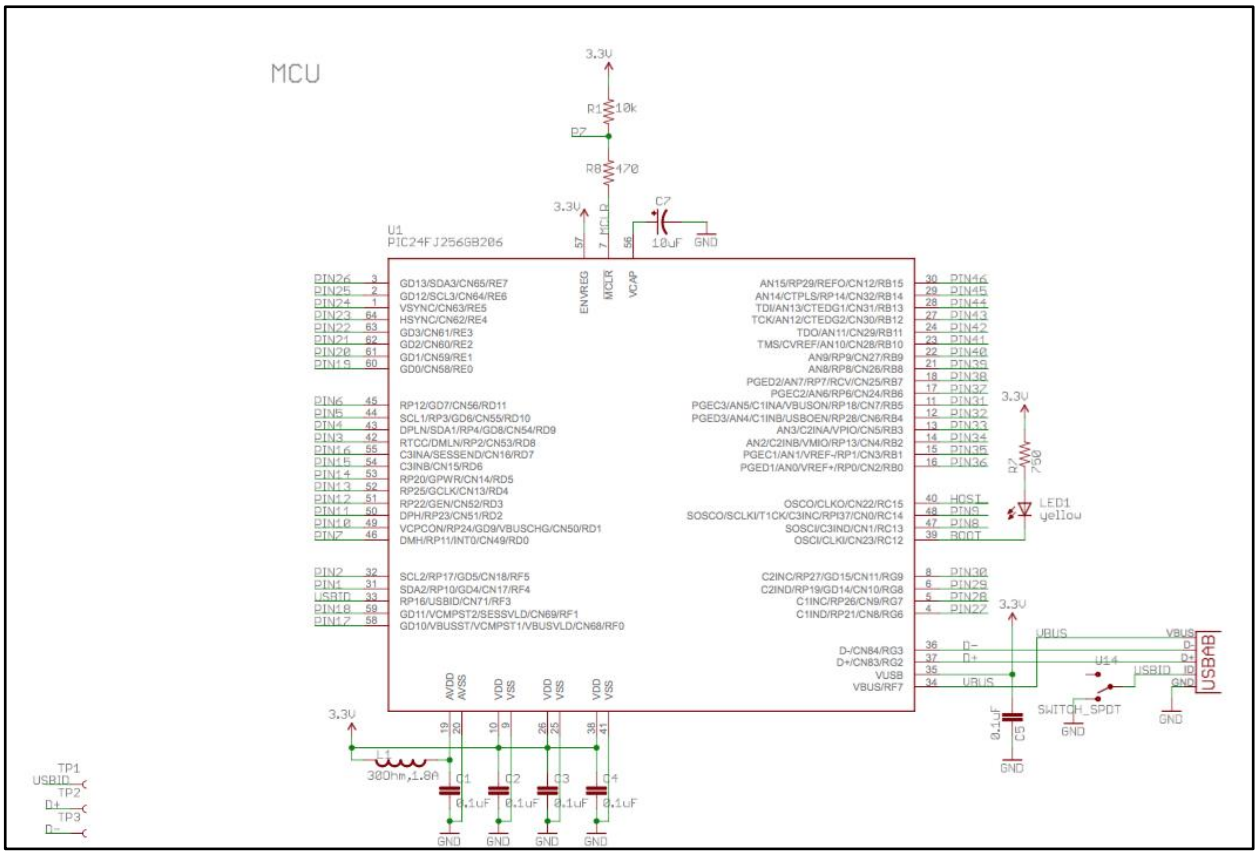


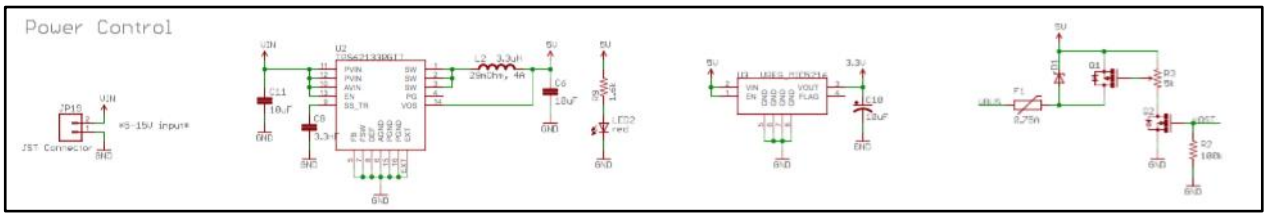
Note 1: The back pad on QFN devices should be connected to Vss.
Legend: **RPn** and **RPIn** represents remappable peripheral pins.
Shaded pins indicate pins that are tolerant to up to +5.5V.

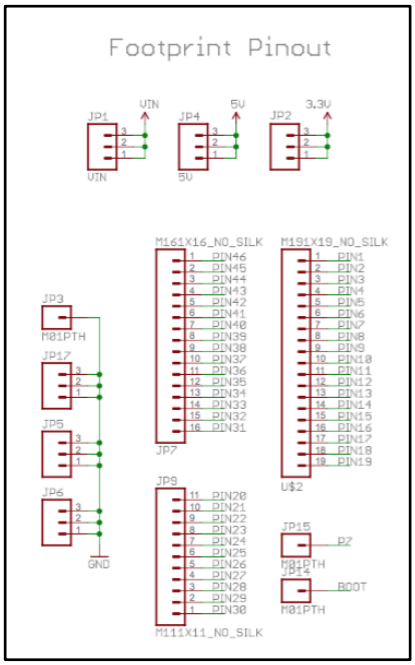
© 2010 Microchip Technology Inc.

2. Esquemático de IOIO

Fuente: [24]







3. Datasheet de los temporizadores de la familia de microcontroladores PIC24F

Fuente: [25]

PIC24F Family Reference Manual

14.1 INTRODUCTION

Depending on the specific variant, the PIC24F device family offers several 16-bit timers. These timers are designated as Timer1, Timer2, Timer3, ..., etc.

Each timer module is a 16-bit timer/counter consisting of the following readable/writable registers:

- TMRx: 16-Bit Timer Count register
- PRx: 16-Bit Timer Period register associated with the timer
- TxCON: 16-Bit Timer Control register associated with the timer

Each timer module also has the associated bits for interrupt control:

- Interrupt Enable Control bit (TxIE)
- Interrupt Flag Status bit (TxIF)
- Interrupt Priority Control bits (TxIP<2:0>)

With certain exceptions, all of the 16-bit timers have the same functional circuitry. The 16-bit timers are classified into three types to account for their functional differences:

- Type A time base
- Type B time base
- Type C time base

Some 16-bit timers can be combined to form a 32-bit timer.

This section does not describe the dedicated timers that are associated with peripheral devices. For example, this includes the time base associated with the input capture or output compare modules.

 © 2006 Microchip Technology Inc.

4. Datasheet del convertidor A/D de la familia de microcontroladores PIC24F

Fuente: [26]

PIC24F Family Reference Manual

17.1 INTRODUCTION

The PIC24F 10-bit A/D Converter has the following key features:

- Successive Approximation Register (SAR) Conversion
- Conversion Speeds of up to 500 ksps
- Up to 16 External Analog Input Channels
- Multiple Internal Reference Input Channels (select devices only)
- External Voltage Reference Input Pins
- Unipolar Differential Sample-and-Hold (S/H) Amplifier
- Automatic Channel Scan mode
- Selectable Conversion Trigger Source
- 16-Word Conversion Result Buffer
- Selectable Buffer Fill modes
- Four Options for Results Alignment
- Operation during CPU Sleep and Idle modes

The 10-bit A/D Converter module accepts a single analog signal at any one instant and converts it to a corresponding 10-bit digital value. It accomodates up to 16 analog inputs and separate reference inputs; the actual number available on a particular device depends on the package size. The heart of the module is a Successive Approximation Register (SAR) type of A/D Converter. Hardware features surrounding the SAR provide flexible configuration and hardware support for automatic operation, and minimize software overhead, especially in high-speed operation. The three major sections surrounding the ADC are analog input selection, a memory mapped output buffer, and timing and control functions.

An internal Sample-and-Hold (S/H) amplifier acquires a sample of an input signal, then holds that value constant during the conversion process. A combination of input multiplexers selects the signal to be converted from multiple analog input pins. The whole multiplexer path includes provision for differential analog input, although the number of negative input pins is limited, and the signal difference must remain positive (i.e., unipolar). The sampled voltage is held and converted to a digital value, which strictly speaking, represents the ratio of that input voltage to a reference voltage. Configuration choices allow connection of an external reference or use of the device power and ground (AV_{DD} and AV_{SS}). Reference and input signal pins are assigned differently depending on the particular device.

An array of timing and control selections allow the user to create flexible scanning sequences. Conversions can be started individually by program control, continuously free running, or triggered by selected hardware events. A single channel may be repeatedly converted; alternate conversions may be performed on two channels, or any or all of the channels may be sequentially scanned and converted according to a user-defined bit map. The resulting conversion output is a 10-bit digital number which can be signed or unsigned, left or right justified.

Conversions are automatically stored in a dedicated 16-word buffer, allowing for multiple successive readings to be taken before software service is needed. Successive conversions are placed into sequential buffer locations. Alternatively, the buffer can be split into two 8-word sections for simultaneous conversion and read operations. The module sets its interrupt flag after a selectable number of conversions, from one to sixteen, when the whole buffer can be read. After the interrupt, the sequence restarts at the beginning of the buffer. When the interrupt flag is set according to the earlier selection, scan selections and the Output Buffer Pointer return to their starting positions.

A simplified block diagram for the module is shown in Figure 17-1.

© 2009 Microchip Technology Inc.

ANEXO B: CÓDIGO FUENTE

Este anexo incluye el código fuente modificado del firmware y la librería de IOIO, así como la aplicación creada para el sistema de monitorización de señales. Para consultar el código completo deberá dirigirse al archivo comprimido que acompaña a este documento.

1. Firmware

program_tmr1.h

```
// Implements Timer1-related functions.
// Usage:
// setTMR1(ms);
// ...set TMR1 registers in order to enable a TMR1 interrupt every ms
milliseconds.
// enableTMR1();
// ...from now on will send periodic interrupts every ms milliseconds...
// disableTMR1();
// ...will no longer enable TMR1 interrupts...
// setTMR1IF();
// ...will change the Interruption Flag value of Timer1.

#ifndef PROGRAM_TMR1_H
#define     PROGRAM_TMR1_H

// Set TMR1 registers
void setTMR1(unsigned int ms);

// Enable TMR1 Timer
void enableTMR1();

// Disable TMR1 Timer
void disableTMR1();

// Set TMR1 interruption flag
void setTMR1IF(unsigned TMR1IF);

#endif      /* PROGRAM_TMR1_H */
```

program_tmr1.c

```
#include "program_tmr1.h"
#include "Compiler.h"

// Set TMR1 registers to generate an interruption
// every ms milliseconds.
// T=4/64Mhz*(2^16-COMP)*Preescaler
void setTMR1(unsigned int ms) {
    TMR1 = 0x0000;
    if (ms < 4) {
        // 00 = 1:1 prescaler
        T1CON = 0x0000;
        PR1 = (unsigned int)(((double)ms*64000)/4);
    }
    if(ms > 4 && ms < 32) {
        // 01 = 1:8 prescaler
        T1CON = 0x0010;
        PR1 = (unsigned int)(((double)ms*64000)/(4*8));
    }
    if(ms > 33 && ms < 262) {
        // 10 = 1:64 prescaler
        T1CON = 0x0020;
        PR1 = (unsigned int)(((double)ms*64000)/(4*64));
    }
    if(ms > 263) {
        // Up to 1048 ms
        // 11 = 1:256 prescaler
        T1CON = 0x0030;
        PR1 = (unsigned int)(((double)ms*64000)/(4*256));
    }

    IPC0bits.T1IP = 0x01;   // Interruption priority
    IFS0bits.T1IF = 0;      // Interruption flag
    IEC0bits.T1IE = 1;      // Interruptions enabled
}

void enableTMR1() {
    T1CONbits.TON = 1;
}

void disableTMR1() {
    T1CONbits.TON = 0;
}

void setTMR1IF(unsigned TMR1IF) {
    IFS0bits.T1IF = TMR1IF;

}
```

adc_offline.h

```
// Implements A/D offline-related functions of the protocol.
// Usage:
// ADCOfflineSetScan();
// ...from now on will send periodic samples of pin 36...
// ADCOfflineStop();
// ...will no longer send samples...
// ReportADCResults();
// ...will send the packets with the samples.

#ifndef ADC_OFFLINE_H
#define     ADC_OFFLINE_H

// Initialize this module.
// Can be used any time to reset the module's state.
void ADCOfflineSetScan();

// Will send saved samples.
void ReportADCResults();

// Will stop sampling on pin 36.
void ADCOfflineStop();

#endif      /* ADC_OFFLINE_H */
```

adc_offline.c

```
#include "adc_offline.h"

#include <assert.h>
#include <stdint.h>
#include <stdbool.h>
#include <libpic30.h>


#include "Compiler.h"
#include "logging.h"
#include "protocol.h"
#include "pins.h"
#include "sync.h"
#include "program_tmr1.h"
#include "features.h"


volatile unsigned int* buf = &ADC1BUF0;                 // ADC buffer
volatile unsigned int ADCResult[NUM_SAMPLES*NUM_MSG];   // Sampling array
volatile int samplingIndex=0;                           // Number of samples
read
volatile int numPaq=-1;                                 // Inactive state
volatile int totalPaq=0;                                // Number of paquets
to send

static inline void ADCOfflineStart() {
    // Clear any possibly remaining interrupts before enabling them.
    _AD1IF = 0;

    // We can enable interrupts now, they won't fire.
```

```
    _AD1IE = 1;

    // Initializing buffers
    int i;
    for(i=0; i<NUM_SAMPLES*NUM_MSG;i++) {
        ADCResult[i]=0;
    }
    samplingIndex=0;
    numPaq=-1;
    totalPaq=0;

    // Initiliaze the timer. Stopped if already running.
    setTMR1(1000);
    enableTMR1();
}


void ADCOfflineStop() {
    // Disable interrupts. T1 must comes last, as the handlers of the others
may
    // enable it.
    _AD1IE = 0;
    _T1IE = 0;
    // No more interrupts at this point.
    _ADON = 0;    // ADC off
}

void ADCOfflineSetScan() {
    SetPinDigitalOut(0, 1, 1);
    ADCOfflineStop();     // Just in case we were running. No interrupts
after this point.

    // Now nothing will interrupt us
    AD1CON1 = 0x0000;  // ADC off.
    AD1CON2 = 0x0000;  // Avdd Avss ref, single buffer, interrupt on every
sample
    AD1CON3 = 0x1F01;  // * system clock, 31 Tad acquisition time, ADC clock
@8MHz
    AD1CHS  = 0x0000;  // * Sample AN0 against negative reference.
    AD1CSSL = 0x0001;  // reset scan mask, scan channel 0 (pin 36)

    ADCOfflineStart();
}

void ADCOfflineTrigger() {
    _SMPI = 0;
    _SSRC = 7;  // auto-convert
    _CSCNA = 1; // scan channels set in AD1CSSL

    // Turn the module on and start and automatic (scan) sample
    _ADON = 1;
    _ASAM = 1;
}

void ReportADCResults() {
    int i=0;
    disableTMR1();
    SetPinDigitalOut(0, 1, 1);
    OUTGOING_MESSAGE msg;
    msg.type=REPORT_ANALOG_OFFLINE_STATUS;
    /**
     *  Case 1: First packet
```

```
     */
    if(numPaq==-1) {
        numPaq = samplingIndex/NUM_SAMPLES;
        if (samplingIndex % NUM_SAMPLES > 0) {
            numPaq+=1;
        }
        totalPaq=numPaq;        // Number of packets to send
        msg.args.report_analog_offline_status.numMsg=numPaq;
        /**
         *  Case 1.1: Number of registered samples fewer than packet size
         */
        if(samplingIndex<NUM_SAMPLES) {
            msg.args.report_analog_offline_status.numSamples=samplingIndex;
            for(i=0; i<samplingIndex;i++) {

msg.args.report_analog_offline_status.ADCResult[i]=ADCResult[i];
            }
        }
        /**
         *  Case 1.2: Number of registered samples longer than packet size
         */
        else {
            msg.args.report_analog_offline_status.numSamples=NUM_SAMPLES;
            for(i=0; i<NUM_SAMPLES;i++) {

msg.args.report_analog_offline_status.ADCResult[i]=ADCResult[i];
            }
        }
    }
    /**
     *  Case 2: Last packet
     */
    else if (numPaq==2) {
        numPaq--;
        msg.args.report_analog_offline_status.numMsg=numPaq;
        /**
         *  Case 2.1: Number of registered samples equal to packet size
         */
        if(samplingIndex%NUM_SAMPLES==0) {
            msg.args.report_analog_offline_status.numSamples=NUM_SAMPLES;
            for(i=(totalPaq-numPaq)*NUM_SAMPLES; i<(totalPaq-
numPaq+1)*NUM_SAMPLES;i++) {
                msg.args.report_analog_offline_status.ADCResult[i-(totalPaq-
numPaq)*NUM_SAMPLES]=ADCResult[i];
            }
        }
        /**
         *  Case 2.2: Number of registered samples fewer than packet size
         */
        else {
            msg.args.report_analog_offline_status.numSamples=samplingIndex %
NUM_SAMPLES;
            for(i=(totalPaq-numPaq)*NUM_SAMPLES; i<samplingIndex;i++) {
                msg.args.report_analog_offline_status.ADCResult[i-(totalPaq-
numPaq)*NUM_SAMPLES]=ADCResult[i];
            }
        }
    }
    /**
     *  Case 3: Intermediate packets
     */
    else {
```

```
        numPaq--;
        msg.args.report_analog_offline_status.numMsg=numPaq;
        msg.args.report_analog_offline_status.numSamples=NUM_SAMPLES;
        for(i=(totalPaq-numPaq)*NUM_SAMPLES; i<(totalPaq-
numPaq+1)*NUM_SAMPLES;i++) {
            msg.args.report_analog_offline_status.ADCResult[i-(totalPaq-
numPaq)*NUM_SAMPLES]=ADCResult[i];
        }
    }

    AppProtocolSendMessage(&msg);
}

/**
 * Timer1 Interrupt Service Routine (ISR).
 * When an interrupt is received, an ADC is triggered.
 */
void __attribute__((__interrupt__, no_auto_psv)) _T1Interrupt(void) {
    ADCOfflineTrigger();

    setTMR1IF(0);
}
```

adc.c

```
#include "adc.h"

#include <assert.h>
#include <stdint.h>
#include <stdbool.h>
#include <libpic30.h>


#include "Compiler.h"
#include "logging.h"
#include "protocol.h"
#include "pins.h"
#include "sync.h"
#include "adc_offline.h"
#include "program_tmr1.h"

#define FAST_INT_PRIORITY 6
#define SLOW_INT_PRIORITY 1

extern int ADCtype;
extern int ADCResult[NUM_SAMPLES];
extern int samplingIndex;

static volatile uint16_t analog_scan_bitmask;
static volatile int analog_scan_num_channels;
// Bit k is set iff channel k is marked for capsense reporting.
static volatile uint16_t capsense_bitmask;
// Bit k is set iff the status of channel k has changed wrt capsense sampling
// and we have not yet reported back this fact. This bit is set whenever the
// status changes and cleared as soon as we report.
static volatile uint16_t capsense_dirty_bitmask;
// Current channel to capsense.
static uint16_t capsense_current = 15;
```

```
// Set to true before triggering a sample to designate this is a cap-sense
// sample.
static bool capsense_sample = false;
// Used to decide whether or not to enable T3 interrupt. When 0, interrupt
// should be enabled, otherwise, disabled.
static volatile int t3_int_counter;

// we need to generate a priority 1 interrupt in order to send a message
// containing ADC-captured data.
// this is the reasononing:
// we need to protect the outgoing-message buffer from concurrent access.
this
// is achieved by making sure it is only written to by priority 1 code.
// however, in the case of ADC, we must service the "done" interrupt quickly
// to stop the ADC before our buffer gets overwritten, so this would be a
// priority 6 interrupt. then, in order to write to the output buffer, it
would
// trigger the priority 1 interrupt using GFX1, that will read the ADC data
and
// write to the buffer.
// we've "abused" CRC interrupt that is never used in order to generate the
// interrupt - we just manually raise its IF flag whenever we need an
interrupt
// and service this interrupt.
static inline void ScanDoneInterruptInit() {
  _CRCIP = SLOW_INT_PRIORITY;
}

// call this function to generate the priority 1 interrupt.
static inline void ScanDoneInterruptTrigger() {
  _CRCIF = 1;
}

// timer 3 is clocked @2MHz
// we set its period to 2000 so that a match occurs @1KHz
// used for ADC
static inline void Timer3Init() {
  PR3   = 1999;  // period is 2000 clocks = 1KHz
  _T3IP = SLOW_INT_PRIORITY; // interrupt priority 1 (this interrupt may
write to outgoing channel)
}

static inline void T3IntBlock() {
  PRIORITY(1) {
    _T3IE = 0;
    ++t3_int_counter;
  }
}

static inline void T3IntUnblock() {
  PRIORITY(1) {
    if (--t3_int_counter == 0) {
      _T3IE = 1;
    }
  }
}

static inline void ADCStart() {
  // Clear any possibly remaining interrupts before enabling them.
  _AD1IF = 0;
  _CRCIF = 0;
```

```
  _CRCIE = 1;  // We can enable interrupts now, they won't fire.
  _AD1IE = 1;

  t3_int_counter = 0;
  // Reset counter and start triggering.
  TMR3  = 0x0000;  // reset counter
  _T3IF = 0;
  _T3IE = 1;  // We're ready to handle trigger interrupts.
}

// IMPORTANT:
// A post-condition of this function is that no interrupts (related to this
// module) will fire.
static inline void ADCStop() {
  // Disable interrupts. T3 must comes last, as the handlers of the others
may
  // enable it.
  _AD1IE = 0;
  _CRCIE = 0;
  _T3IE = 0;
  // No more interrupts at this point.
  _CTMUEN = 0;  // CTMU off.
  _ADON = 0;    // ADC off
}

void ADCInit() {
  ADCStop();     // Just in case we were running. No interrupts after this
point.
  Timer3Init();  // initiliaze the timer. stopped if already running.

  // Now nothing will interrupt us
  AD1CON1 = 0x0000;  // ADC off.
  AD1CON2 = 0x0000;  // Avdd Avss ref, single buffer, interrupt on every
sample
  AD1CON3 = 0x1F01;  // system clock, 31 Tad acquisition time, ADC clock
@8MHz
  AD1CHS  = 0x0000;  // Sample AN0 against negative reference.
  AD1CSSL = 0x0000;  // reset scan mask.

  _AD1IP = FAST_INT_PRIORITY;        // high priority to stop automatic
sampling

  // Setup CTMU
  CTMUCON = (1 << 8)  // CTTRIG
            | (0x3 << 5) // EDG2SEL
            | (0x3 << 2) // EDG1SEL
            | (1 << 4);  // EDG1POL;
  CTMUICON = 0x7F00; //89.1uA

  ScanDoneInterruptInit();  // when triggered, generates an immediate
interrupt to read ADC buffer

  analog_scan_bitmask = 0x0000;
  analog_scan_num_channels = 0;
  capsense_bitmask = 0x0000;
  capsense_dirty_bitmask = 0x0000;
}

static inline int CountOnes(unsigned int val) {
  int res = 0;
  while (val) {
    if (val & 1) ++res;
```

```
    val >>= 1;
  }
  return res;
}

static inline void ReportAnalogInStatus() {
  volatile unsigned int* buf = &ADC1BUF0;
  int num_channels = CountOnes(AD1CSSL);
  int i;
  BYTE var_arg[16 / 4 * 5];
  int var_arg_pos = 0;
  int group_header_pos;
  int pos_in_group;
  int value;
  OUTGOING_MESSAGE msg;
  msg.type = REPORT_ANALOG_IN_STATUS;
  for (i = 0; i < num_channels; i++) {
    pos_in_group = i & 3;
    if (pos_in_group == 0) {
      group_header_pos = var_arg_pos;
      var_arg[var_arg_pos++] = 0;  // reset header
    }
    value = buf[i];
    //log_printf("%d", value);
    var_arg[group_header_pos] |= (value & 3) << (pos_in_group * 2);  // two
LSb to group header
    var_arg[var_arg_pos++] = value >> 2;  // eight MSb to channel byte
  }
  AppProtocolSendMessageWithVarArg(&msg, var_arg, var_arg_pos);
}

static inline void ReportCapSense() {
  OUTGOING_MESSAGE msg;
  msg.type = CAPSENSE_REPORT;
  msg.args.capsense_report.pin = PinFromAnalogChannel(capsense_current);
  msg.args.capsense_report.value = ADC1BUF0;
  AppProtocolSendMessage(&msg);
}

static inline void ReportAnalogInFormat() {
  unsigned int mask = analog_scan_bitmask;
  int channel = 0;
  BYTE var_arg[16];
  int var_arg_pos = 0;
  OUTGOING_MESSAGE msg;
  msg.type = REPORT_ANALOG_IN_FORMAT;
  msg.args.report_analog_in_format.num_pins = analog_scan_num_channels;
  while (mask) {
    if (mask & 1) {
      var_arg[var_arg_pos++] = PinFromAnalogChannel(channel);
    }
    mask >>= 1;
    ++channel;
  }
  AppProtocolSendMessageWithVarArg(&msg, var_arg, var_arg_pos);
  AD1CSSL = analog_scan_bitmask;
}


static inline void ReportModifiedCapSenseStatus() {
  int i = 0;
  OUTGOING_MESSAGE msg;
```

```
  msg.type = SET_CAPSENSE_SAMPLING;

  uint16_t mask = capsense_bitmask;  // Local copy.
  while (capsense_dirty_bitmask) {
    if (capsense_dirty_bitmask & 1) {
      msg.args.set_capsense_sampling.pin = PinFromAnalogChannel(i);
      msg.args.set_capsense_sampling.enable = mask & 1;
      AppProtocolSendMessage(&msg);
    }
    capsense_dirty_bitmask >>= 1;
    mask >>= 1;
    ++i;
  }
}

static inline void ADCTrigger() {
  assert(analog_scan_num_channels != 0);

  _SMPI = analog_scan_num_channels - 1;
  _SSRC = 7;  // auto-convert.
  _CSCNA = 1; // scan channels set in AD1CSSL
  capsense_sample = false;  // let the ISR know this is an analog scan.

  // Turn the module on and start and automatic (scan) sample.
  _ADON = 1;
  _ASAM = 1;
}

static inline void ADCCapSenseTrigger() {
  assert(capsense_bitmask);

  // Cycle...
  capsense_current = (capsense_current + 1) & 0x0F;

  if ((1 << capsense_current) & capsense_bitmask) {
    _CTMUEN = 1;  // CTMU on.
    _IDISSEN = 1;  // discharge ADC internal cap.
    _CH0SA = capsense_current;  // select channel to sample.
    _SMPI = 0;     // interrupt when done.
    _SSRC = 6;     // convert once CTMU current pulse is done.
    _CSCNA = 0;    // don't scan.
    capsense_sample = true;  // let the ISR know this is a capsense sample.
    // Stop discharging circuit.
    PinSetTris(PinFromAnalogChannel(capsense_current), 1);
    _IDISSEN = 0;  // Stop discharging internal cap.

    // Turn the module on and start manual sampling.
    _ADON = 1;
    _SAMP = 1;

    // Charge for 16 cycles (1us) at constant current.
    _EDG1STAT = 1; // Set edge1 - Start Charge
    __delay32(15);
    _EDG1STAT = 0; //Clear edge1 - Stop Charge - auto-triggers ADC
conversion.
  } else {
    T3IntUnblock();
  }
}

void ADCSetScan(int pin, int enable) {
  log_printf("ADCSetScan(%d, %d)", pin, enable);
```

```
  int channel = PinToAnalogChannel(pin);
  int mask;
  if (channel == -1) return;
  mask = 1 << channel;
  if (!!(mask & analog_scan_bitmask) == enable) return;

  if (enable) {
    if (analog_scan_bitmask | capsense_bitmask) {
      // already running, just add the new channel
      T3IntBlock();
      // These two variables are shared with the triggering code, ran from
      // timer 3 interrupt context.
      ++analog_scan_num_channels;
      analog_scan_bitmask |= mask;
      T3IntUnblock();
    } else {
      // first channel, start running
      analog_scan_num_channels = 1;
      analog_scan_bitmask = mask;
      ADCStart();
    }
  } else {
    T3IntBlock();
    --analog_scan_num_channels;
    analog_scan_bitmask &= ~mask;
    if (analog_scan_bitmask | capsense_bitmask) {
      T3IntUnblock();
    } else {
      // This was the last channel. At this point no new samples will be
      // triggered, but we may be in the middle of a sample.
      ADCStop();
      // Now we're safe. Report the change in format.
      ReportAnalogInFormat();
    }
  }
}

void ADCSetCapSense(int pin, int enable) {
  log_printf("ADCSetCapSense(%d, %d)", pin, enable);
  int channel = PinToAnalogChannel(pin);
  int mask;
  if (channel == -1) return;
  mask = 1 << channel;
  if (!!(mask & capsense_bitmask) == enable) return;

  if (enable) {
    if (analog_scan_bitmask | capsense_bitmask) {
      // already running, just add the new channel
      T3IntBlock();
      // These two variables are shared with the triggering code, ran from
      // timer 3 interrupt context.
      capsense_bitmask |= mask;
      capsense_dirty_bitmask |= mask;
      T3IntUnblock();
    } else {
      // first channel, start running
      capsense_bitmask = mask;
      capsense_dirty_bitmask = mask;
      ADCStart();
    }
  } else {
    T3IntBlock();
```

```
    capsense_bitmask &= ~mask;
    capsense_dirty_bitmask |= mask;
    if (analog_scan_bitmask | capsense_bitmask) {
      T3IntUnblock();
    } else {
      // This was the last channel. At this point no new samples will be
      // triggered, but we may be in the middle of a sample.
      ADCStop();
      // Now we're safe. Report the change in format.
      ReportModifiedCapSenseStatus();
    }
  }
}

void __attribute__((__interrupt__, auto_psv)) _T3Interrupt() {
  // Report frame format of analog channels if changed.
  if (AD1CSSL != analog_scan_bitmask) {
    ReportAnalogInFormat();
  }
  assert(AD1CSSL == analog_scan_bitmask);

  // Report enabling / disabling of capsense channels.
  if (capsense_dirty_bitmask) {
    ReportModifiedCapSenseStatus();
  }
  assert(!capsense_dirty_bitmask);

  T3IntBlock();  // disable interrupts. will be re-enabled when sampling is
done.
  // Sample!
  if (analog_scan_num_channels) {
    // Trigger ADC sequence, which will eventually trigger capsense.
    ADCTrigger();
  } else if (capsense_bitmask) {
    // Jump directly to capsense.
    ADCCapSenseTrigger();
  } else {
    assert(false);
  }
  _T3IF = 0;  // clear
}

void __attribute__((__interrupt__, auto_psv)) _CRCInterrupt() {
  if (capsense_sample) {
    _CTMUEN = 0; // CTMU off.
    // Discharge circuit.
    PinSetTris(PinFromAnalogChannel(capsense_current), 0);
    ReportCapSense();
    T3IntUnblock();  // ready for next trigger.
  } else {
    ReportAnalogInStatus();
    if (capsense_bitmask) {
      ADCCapSenseTrigger();
    } else {
      T3IntUnblock();  // ready for next trigger.
    }
  }
  _CRCIF = 0;  // clear
}

/**
 * A/D Interrupt Service Routine.
```

```
 * When an interrupt is received, an ADC is saved.
 */
void __attribute__((__interrupt__, auto_psv)) _ADC1Interrupt() {
  led_on();

  // ADC Online
  if (ADCtype==0)
    ScanDoneInterruptTrigger();
  // ADC Offline
  else {
    // Save samples until the array is full.
    if(samplingIndex<NUM_SAMPLES*NUM_MSG) {
        ADCResult[samplingIndex]=ADC1BUF0;
        samplingIndex++;
    }
    //When the array is full, disable Timer1 and ADC
    else if(samplingIndex==NUM_SAMPLES*NUM_MSG) {
        disableTMR1();
        ADCOfflineStop();
    }
  }
  _ADON = 0;  // Turn the module off
  _AD1IF = 0;  // Clear

  // Led blinking during the sampling
  int k=0;
  for(k=0; k<500;k++){}
    led_off();
}
```

protocol_defs.h

```
#ifndef __PROTOCOLDEFS_H__
#define __PROTOCOLDEFS_H__

#include "GenericTypeDefs.h"

#define PACKED __attribute__ ((packed))

#define IOIO_MAGIC 0x4F494F49L

#define PROTOCOL_IID_IOIO0001 "IOIO0001"
#define PROTOCOL_IID_IOIO0002 "IOIO0002"
#define PROTOCOL_IID_IOIO0003 "IOIO0003"
#define PROTOCOL_IID_IOIO0004 "IOIO0004"
#define PROTOCOL_IID_IOIO0005 "IOIO0005"

#define  NUM_SAMPLES   125
#define  NUM_MSG       10

// hard reset
typedef struct PACKED {
  DWORD magic;
} HARD_RESET_ARGS;

// establish connection
typedef struct PACKED {
  DWORD magic;
  BYTE hw_impl_ver[8];
```

```
  BYTE bl_impl_ver[8];
  BYTE fw_impl_ver[8];
} ESTABLISH_CONNECTION_ARGS;

// soft reset
typedef struct PACKED {
} SOFT_RESET_ARGS;

// set pin digital out
typedef struct PACKED {
  BYTE open_drain : 1;
  BYTE value : 1;
  BYTE pin : 6;
} SET_PIN_DIGITAL_OUT_ARGS;

// set digital out level
typedef struct PACKED {
  BYTE value : 1;
  BYTE : 1;
  BYTE pin : 6;
} SET_DIGITAL_OUT_LEVEL_ARGS;

// report digital in status
typedef struct PACKED {
  BYTE level : 1;
  BYTE : 1;
  BYTE pin : 6;
} REPORT_DIGITAL_IN_STATUS_ARGS;

// set pin digital in
typedef struct PACKED {
  BYTE pull : 2;
  BYTE pin : 6;
} SET_PIN_DIGITAL_IN_ARGS;

// set change notify
typedef struct PACKED {
  BYTE cn : 1;
  BYTE : 1;
  BYTE pin : 6;
} SET_CHANGE_NOTIFY_ARGS;

// register periodic digital sampling
typedef struct PACKED {
  BYTE pin : 6;
  BYTE : 2;
  BYTE freq_scale;
} REGISTER_PERIODIC_DIGITAL_SAMPLING_ARGS;

// report periodic digital in status
typedef struct PACKED {
  BYTE size;
} REPORT_PERIODIC_DIGITAL_IN_STATUS_ARGS;

// reserved
typedef struct PACKED {
  // for future use
} RESERVED_ARGS;

// set pin pwm
typedef struct PACKED {
  BYTE pin : 6;
```

```
  BYTE : 2;
  BYTE pwm_num : 4;
  BYTE : 3;
  BYTE enable : 1;
} SET_PIN_PWM_ARGS;

// set pwm duty cycle
typedef struct PACKED {
  BYTE fraction : 2;
  BYTE pwm_num : 4;
  BYTE : 2;
  WORD dc;
} SET_PWM_DUTY_CYCLE_ARGS;

// set pwm period
typedef struct PACKED {
  BYTE scale_l : 1;
  BYTE pwm_num : 4;
  BYTE : 2;
  BYTE scale_h : 1;
  WORD period;
} SET_PWM_PERIOD_ARGS;

// uart report tx status
typedef struct PACKED {
  BYTE uart_num : 2;
  WORD bytes_to_add : 14;
} UART_REPORT_TX_STATUS_ARGS;

// set pin analog in
typedef struct PACKED {
  BYTE pin;
} SET_PIN_ANALOG_IN_ARGS;

// report analog in format
typedef struct PACKED {
  BYTE num_pins;
} REPORT_ANALOG_IN_FORMAT_ARGS;

// report analog in status
typedef struct PACKED {
} REPORT_ANALOG_IN_STATUS_ARGS;

// uart data
typedef struct PACKED {
  BYTE size : 6;
  BYTE uart_num : 2;
  BYTE data[0];
} UART_DATA_ARGS;

// uart config
typedef struct PACKED {
  BYTE parity : 2;
  BYTE two_stop_bits : 1;
  BYTE speed4x : 1;
  BYTE : 2;
  BYTE uart_num : 2;
  WORD rate;
} UART_CONFIG_ARGS;

// uart status
typedef struct PACKED {
```

```
  BYTE uart_num : 2;
  BYTE : 5;
  BYTE enabled : 1;
} UART_STATUS_ARGS;

// set pin uart
typedef struct PACKED {
  BYTE pin : 6;
  BYTE : 2;
  BYTE uart_num : 2;
  BYTE : 4;
  BYTE dir : 1;
  BYTE enable : 1;
} SET_PIN_UART_ARGS;

// spi report tx status
typedef struct PACKED {
  BYTE spi_num : 2;
  WORD bytes_to_add : 14;
} SPI_REPORT_TX_STATUS_ARGS;

// spi data
typedef struct PACKED {
  BYTE size : 6;
  BYTE spi_num : 2;
  BYTE ss_pin : 6;
  BYTE : 2;
  BYTE data[0];
} SPI_DATA_ARGS;

// spi master request
typedef struct PACKED {
  BYTE ss_pin : 6;
  BYTE spi_num : 2;
  BYTE total_size : 6;
  BYTE res_size_neq_total : 1;
  BYTE data_size_neq_total : 1;
  union {
    BYTE data_size;
    BYTE vararg[0];
  };
} SPI_MASTER_REQUEST_ARGS;

// spi configure master
typedef struct PACKED {
  BYTE div : 3;
  BYTE scale : 2;
  BYTE spi_num : 2;
  BYTE : 1;
  BYTE clk_pol : 1;
  BYTE clk_edge : 1;
  BYTE smp_end : 1;
  BYTE : 5;
} SPI_CONFIGURE_MASTER_ARGS;

// spi status
typedef struct PACKED {
  BYTE spi_num : 2;
  BYTE : 5;
  BYTE enabled : 1;
} SPI_STATUS_ARGS;
```

```
// set pin spi
typedef struct PACKED {
  BYTE pin : 6;
  BYTE : 2;
  BYTE spi_num : 2;
  BYTE mode : 2;
  BYTE enable : 1;
  BYTE : 3;
} SET_PIN_SPI_ARGS;

// i2c configure master
typedef struct PACKED {
  BYTE i2c_num : 2;
  BYTE : 3;
  BYTE rate : 2;
  BYTE smbus_levels : 1;
} I2C_CONFIGURE_MASTER_ARGS;

// i2c status
typedef struct PACKED {
  BYTE i2c_num : 2;
  BYTE : 5;
  BYTE enabled : 1;
} I2C_STATUS_ARGS;

// i2c write read
typedef struct PACKED {
  BYTE i2c_num : 2;
  BYTE : 3;
  BYTE ten_bit_addr : 1;
  BYTE addr_msb : 2;
  BYTE addr_lsb;
  BYTE write_size;
  BYTE read_size;
  BYTE data[0];
} I2C_WRITE_READ_ARGS;

// i2c result
typedef struct PACKED {
  BYTE i2c_num : 2;
  BYTE : 6;
  BYTE size;
} I2C_RESULT_ARGS;

// i2c report tx status
typedef struct PACKED {
  BYTE i2c_num : 2;
  WORD bytes_to_add : 14;
} I2C_REPORT_TX_STATUS_ARGS;

// set analog pin sampling
typedef struct PACKED {
  BYTE pin : 6;
  BYTE : 1;
  BYTE enable : 1;
} SET_ANALOG_IN_SAMPLING_ARGS;

// check interface
typedef struct PACKED {
  BYTE interface_id[8];
} CHECK_INTERFACE_ARGS;
```

```
// check interface response
typedef struct PACKED {
  BYTE supported : 1;
  BYTE : 7;
} CHECK_INTERFACE_RESPONSE_ARGS;

// icsp six
typedef struct PACKED {
    DWORD inst : 24;
} ICSP_SIX_ARGS;

// icsp report rx status
typedef struct PACKED {
    WORD bytes_to_add;
} ICSP_REPORT_RX_STATUS_ARGS;

// icsp regout
typedef struct PACKED {
} ICSP_REGOUT_ARGS;

// icsp result
typedef struct PACKED {
    WORD reg;
} ICSP_RESULT_ARGS;

// icsp programming enter
typedef struct PACKED {
} ICSP_PROG_ENTER_ARGS;

// icsp programming exit
typedef struct PACKED {
} ICSP_PROG_EXIT_ARGS;

// icsp configure
typedef struct PACKED {
    BYTE enable : 1;
} ICSP_CONFIG_ARGS;

// incap configure
typedef struct PACKED {
    BYTE incap_num : 4;
    BYTE : 4;
    BYTE clock : 2;
    BYTE : 1;
    BYTE mode : 3;
    BYTE : 1;
    BYTE double_prec : 1;
} INCAP_CONFIG_ARGS;

// incap status
typedef struct PACKED {
    BYTE incap_num : 4;
    BYTE : 3;
    BYTE enabled : 1;
} INCAP_STATUS_ARGS;

// set pin incap
typedef struct PACKED {
    BYTE pin : 6;
    BYTE : 2;
    BYTE incap_num : 4;
    BYTE : 3;
```

```
    BYTE enable : 1;
} SET_PIN_INCAP_ARGS;

// incap report
typedef struct PACKED {
    BYTE incap_num : 4;
    BYTE : 2;
    BYTE size : 2;
} INCAP_REPORT_ARGS;

// soft close
typedef struct PACKED {
} SOFT_CLOSE_ARGS;

// set pin capsense
typedef struct PACKED {
  BYTE pin : 6;
  BYTE : 2;
} SET_PIN_CAPSENSE_ARGS;

// capsense report
typedef struct PACKED {
  BYTE pin : 6;
  WORD value : 10;
} CAPSENSE_REPORT_ARGS;

// set capsense sampling
typedef struct PACKED {
  BYTE pin : 6;
  BYTE : 1;
  BYTE enable : 1;
} SET_CAPSENSE_SAMPLING_ARGS;

// sequencer configure
typedef struct PACKED {
  BYTE size;
  BYTE config[0];
} SEQUENCER_CONFIGURE_ARGS;

// sequencer push cue
typedef struct PACKED {
  WORD time;
  BYTE cue[0];
} SEQUENCER_PUSH_ARGS;

// sequencer control
typedef struct PACKED {
  BYTE cmd;
  union {
    BYTE extra[0];
  };
} SEQUENCER_CONTROL_ARGS;

typedef struct PACKED {
  BYTE event;
} SEQUENCER_EVENT_ARGS;

// report analog offline status
typedef struct PACKED {
    BYTE numMsg;
    BYTE numSamples;
    WORD ADCResult[NUM_SAMPLES];
```

```
} REPORT_ANALOG_OFFLINE_STATUS_ARGS;

// BOOKMARK(add_feature): Add a struct for the new incoming / outgoing
message
// arguments.

typedef struct PACKED {
  BYTE type;
  union PACKED {
    HARD_RESET_ARGS                          hard_reset;
    SOFT_RESET_ARGS                          soft_reset;
    SET_PIN_DIGITAL_OUT_ARGS                 set_pin_digital_out;
    SET_DIGITAL_OUT_LEVEL_ARGS               set_digital_out_level;
    SET_PIN_DIGITAL_IN_ARGS                  set_pin_digital_in;
    SET_CHANGE_NOTIFY_ARGS                   set_change_notify;
    REGISTER_PERIODIC_DIGITAL_SAMPLING_ARGS
register_periodic_digital_sampling;
    SET_PIN_PWM_ARGS                         set_pin_pwm;
    SET_PWM_DUTY_CYCLE_ARGS                  set_pwm_duty_cycle;
    SET_PWM_PERIOD_ARGS                      set_pwm_period;
    SET_PIN_ANALOG_IN_ARGS                   set_pin_analog_in;
    UART_DATA_ARGS                           uart_data;
    UART_CONFIG_ARGS                         uart_config;
    SET_PIN_UART_ARGS                        set_pin_uart;
    SPI_MASTER_REQUEST_ARGS                  spi_master_request;
    SPI_CONFIGURE_MASTER_ARGS                spi_configure_master;
    SET_PIN_SPI_ARGS                         set_pin_spi;
    I2C_CONFIGURE_MASTER_ARGS                i2c_configure_master;
    I2C_WRITE_READ_ARGS                      i2c_write_read;
    SET_ANALOG_IN_SAMPLING_ARGS              set_analog_pin_sampling;
    CHECK_INTERFACE_ARGS                     check_interface;
    ICSP_SIX_ARGS                            icsp_six;
    ICSP_REGOUT_ARGS                         icsp_regout;
    ICSP_PROG_ENTER_ARGS                     icsp_prog_enter;
    ICSP_PROG_EXIT_ARGS                      icsp_prog_exit;
    ICSP_CONFIG_ARGS                         icsp_config;
    INCAP_CONFIG_ARGS                        incap_config;
    SET_PIN_INCAP_ARGS                       set_pin_incap;
    SOFT_CLOSE_ARGS                          soft_close;
    SET_PIN_CAPSENSE_ARGS                    set_pin_capsense;
    SET_CAPSENSE_SAMPLING_ARGS               set_capsense_sampling;
    SEQUENCER_CONFIGURE_ARGS                 sequencer_configure;
    SEQUENCER_PUSH_ARGS                      sequencer_push;
    SEQUENCER_CONTROL_ARGS                   sequencer_control;

    // BOOKMARK(add_feature): Add argument struct to the union.
  } args;
  BYTE __vabuf[72];  // buffer for var args. never access directly!
} INCOMING_MESSAGE;

typedef struct PACKED {
  BYTE type;
  union PACKED {
    ESTABLISH_CONNECTION_ARGS              establish_connection;
    REPORT_DIGITAL_IN_STATUS_ARGS          report_digital_in_status;
    REPORT_PERIODIC_DIGITAL_IN_STATUS_ARGS
report_periodic_digital_in_status;
    REPORT_ANALOG_IN_FORMAT_ARGS           report_analog_in_format;
    REPORT_ANALOG_IN_STATUS_ARGS           report_analog_in_status;
    UART_REPORT_TX_STATUS_ARGS             uart_report_tx_status;
    UART_DATA_ARGS                         uart_data;
    SPI_REPORT_TX_STATUS_ARGS              spi_report_tx_status;
```

```
    SPI_DATA_ARGS                           spi_data;
    I2C_RESULT_ARGS                         i2c_result;
    I2C_REPORT_TX_STATUS_ARGS               i2c_report_tx_status;
    CHECK_INTERFACE_RESPONSE_ARGS           check_interface_response;
    UART_STATUS_ARGS                        uart_status;
    SPI_STATUS_ARGS                         spi_status;
    I2C_STATUS_ARGS                         i2c_status;
    ICSP_RESULT_ARGS                        icsp_result;
    ICSP_REPORT_RX_STATUS_ARGS              icsp_report_rx_status;
    INCAP_STATUS_ARGS                       incap_status;
    INCAP_REPORT_ARGS                       incap_report;
    SOFT_CLOSE_ARGS                         soft_close;
    CAPSENSE_REPORT_ARGS                    capsense_report;
    SET_CAPSENSE_SAMPLING_ARGS              set_capsense_sampling;
    SEQUENCER_EVENT_ARGS                    sequencer_event;
    REPORT_ANALOG_OFFLINE_STATUS_ARGS       report_analog_offline_status;
    // BOOKMARK(add_feature): Add argument struct to the union.
  } args;
} OUTGOING_MESSAGE;


typedef enum {
  HARD_RESET                          = 0x00,
  ESTABLISH_CONNECTION                = 0x00,
  SOFT_RESET                          = 0x01,
  CHECK_INTERFACE                     = 0x02,
  CHECK_INTERFACE_RESPONSE            = 0x02,

  SET_PIN_DIGITAL_OUT                 = 0x03,
  SET_DIGITAL_OUT_LEVEL               = 0x04,
  REPORT_DIGITAL_IN_STATUS            = 0x04,
  SET_PIN_DIGITAL_IN                  = 0x05,
  REPORT_PERIODIC_DIGITAL_IN_STATUS   = 0x05,
  SET_CHANGE_NOTIFY                   = 0x06,
  REGISTER_PERIODIC_DIGITAL_SAMPLING  = 0x07,

  SET_PIN_PWM                         = 0x08,
  SET_PWM_DUTY_CYCLE                  = 0x09,
  SET_PWM_PERIOD                      = 0x0A,

  SET_PIN_ANALOG_IN                   = 0x0B,
  REPORT_ANALOG_IN_STATUS             = 0x0B,
  SET_ANALOG_IN_SAMPLING              = 0x0C,
  REPORT_ANALOG_IN_FORMAT             = 0x0C,

  UART_CONFIG                         = 0x0D,
  UART_STATUS                         = 0x0D,
  UART_DATA                           = 0x0E,
  SET_PIN_UART                        = 0x0F,
  UART_REPORT_TX_STATUS               = 0x0F,

  SPI_CONFIGURE_MASTER                = 0x10,
  SPI_STATUS                          = 0x10,
  SPI_MASTER_REQUEST                  = 0x11,
  SPI_DATA                            = 0x11,
  SET_PIN_SPI                         = 0x12,
  SPI_REPORT_TX_STATUS                = 0x12,

  I2C_CONFIGURE_MASTER                = 0x13,
  I2C_STATUS                          = 0x13,
  I2C_WRITE_READ                      = 0x14,
  I2C_RESULT                          = 0x14,
```

```
  I2C_REPORT_TX_STATUS                = 0x15,

  ICSP_SIX                            = 0x16,
  ICSP_REPORT_RX_STATUS               = 0x16,
  ICSP_REGOUT                         = 0x17,
  ICSP_RESULT                         = 0x17,
  ICSP_PROG_ENTER                     = 0x18,
  ICSP_PROG_EXIT                      = 0x19,
  ICSP_CONFIG                         = 0x1A,

  INCAP_CONFIG                        = 0x1B,
  INCAP_STATUS                        = 0x1B,
  SET_PIN_INCAP                       = 0x1C,
  INCAP_REPORT                        = 0x1C,

  SOFT_CLOSE                          = 0x1D,

  SET_PIN_CAPSENSE                    = 0x1E,
  CAPSENSE_REPORT                     = 0x1E,
  SET_CAPSENSE_SAMPLING               = 0x1F,

  SEQUENCER_CONFIGURE                 = 0x20,
  SEQUENCER_EVENT                     = 0x20,
  SEQUENCER_PUSH                      = 0x21,
  SEQUENCER_CONTROL                   = 0x22,

  SYNC                                = 0x23,
  SET_ANALOG_OFFLINE                  = 0x24,
  SET_RETURN_ANALOG_OFFLINE_STATUS    = 0x25,
  REPORT_ANALOG_OFFLINE_STATUS        = 0x26,

  // BOOKMARK(add_feature): Add new message type to enum.
  MESSAGE_TYPE_LIMIT
} MESSAGE_TYPE;
#endif  // __PROTOCOLDEFS_H__
```

protocol.c

```
#include "protocol.h"

#include <assert.h>
#include <string.h>
#include "libpic30.h"
#include "blapi/version.h"
#include "byte_queue.h"
#include "features.h"
#include "pwm.h"
#include "adc.h"
#include "digital.h"
#include "logging.h"
#include "platform.h"
#include "uart.h"
#include "spi.h"
#include "i2c.h"
#include "sync.h"
#include "icsp.h"
#include "incap.h"
#include "sequencer_protocol.h"
```

```
// New include
#include "adc_offline.h"
#include "program_tmr1.h"

int ADCtype = 0;    // 1 if ADC Offline

#define CHECK(cond) do { if (!(cond)) { log_printf("Check failed: %s",
#cond); return FALSE; }} while(0)

const BYTE incoming_arg_size[MESSAGE_TYPE_LIMIT] = {
  sizeof(HARD_RESET_ARGS),
  sizeof(SOFT_RESET_ARGS),
  sizeof(CHECK_INTERFACE_ARGS),
  sizeof(SET_PIN_DIGITAL_OUT_ARGS),
  sizeof(SET_DIGITAL_OUT_LEVEL_ARGS),
  sizeof(SET_PIN_DIGITAL_IN_ARGS),
  sizeof(SET_CHANGE_NOTIFY_ARGS),
  sizeof(REGISTER_PERIODIC_DIGITAL_SAMPLING_ARGS),
  sizeof(SET_PIN_PWM_ARGS),
  sizeof(SET_PWM_DUTY_CYCLE_ARGS),
  sizeof(SET_PWM_PERIOD_ARGS),
  sizeof(SET_PIN_ANALOG_IN_ARGS),
  sizeof(SET_ANALOG_IN_SAMPLING_ARGS),
  sizeof(UART_CONFIG_ARGS),
  sizeof(UART_DATA_ARGS),
  sizeof(SET_PIN_UART_ARGS),
  sizeof(SPI_CONFIGURE_MASTER_ARGS),
  sizeof(SPI_MASTER_REQUEST_ARGS),
  sizeof(SET_PIN_SPI_ARGS),
  sizeof(I2C_CONFIGURE_MASTER_ARGS),
  sizeof(I2C_WRITE_READ_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(ICSP_SIX_ARGS),
  sizeof(ICSP_REGOUT_ARGS),
  sizeof(ICSP_PROG_ENTER_ARGS),
  sizeof(ICSP_PROG_EXIT_ARGS),
  sizeof(ICSP_CONFIG_ARGS),
  sizeof(INCAP_CONFIG_ARGS),
  sizeof(SET_PIN_INCAP_ARGS),
  sizeof(SOFT_CLOSE_ARGS),
  sizeof(SET_PIN_CAPSENSE_ARGS),
  sizeof(SET_CAPSENSE_SAMPLING_ARGS),
  sizeof(SEQUENCER_CONFIGURE_ARGS),
  sizeof(SEQUENCER_PUSH_ARGS),
  sizeof(SEQUENCER_CONTROL_ARGS),
  sizeof(RESERVED_ARGS)

  // BOOKMARK(add_feature): Add sizeof (argument for incoming message).
  // Array is indexed by message type enum.
};

const BYTE outgoing_arg_size[MESSAGE_TYPE_LIMIT] = {
  sizeof(ESTABLISH_CONNECTION_ARGS),
  sizeof(SOFT_RESET_ARGS),
  sizeof(CHECK_INTERFACE_RESPONSE_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(REPORT_DIGITAL_IN_STATUS_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(SET_CHANGE_NOTIFY_ARGS),
  sizeof(REGISTER_PERIODIC_DIGITAL_SAMPLING_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(RESERVED_ARGS),
```

```
  sizeof(RESERVED_ARGS),
  sizeof(REPORT_ANALOG_IN_STATUS_ARGS),
  sizeof(REPORT_ANALOG_IN_FORMAT_ARGS),
  sizeof(UART_STATUS_ARGS),
  sizeof(UART_DATA_ARGS),
  sizeof(UART_REPORT_TX_STATUS_ARGS),
  sizeof(SPI_STATUS_ARGS),
  sizeof(SPI_DATA_ARGS),
  sizeof(SPI_REPORT_TX_STATUS_ARGS),
  sizeof(I2C_STATUS_ARGS),
  sizeof(I2C_RESULT_ARGS),
  sizeof(I2C_REPORT_TX_STATUS_ARGS),
  sizeof(ICSP_REPORT_RX_STATUS_ARGS),
  sizeof(ICSP_RESULT_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(ICSP_CONFIG_ARGS),
  sizeof(INCAP_STATUS_ARGS),
  sizeof(INCAP_REPORT_ARGS),
  sizeof(SOFT_CLOSE_ARGS),
  sizeof(CAPSENSE_REPORT_ARGS),
  sizeof(SET_CAPSENSE_SAMPLING_ARGS),
  sizeof(SEQUENCER_EVENT_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(RESERVED_ARGS),
  sizeof(REPORT_ANALOG_OFFLINE_STATUS_ARGS)
  // BOOKMARK(add_feature): Add sizeof (argument for outgoing message).
  // Array is indexed by message type enum.
};

typedef enum {
  STATE_OPEN,
  STATE_CLOSING,
  STATE_CLOSED
} STATE;

// Not enough RAM in the 24K RAM (old prototypes) platforms, since the
// introduction of the motion control library.
#ifdef __PIC24FJ128DA106__
#define QUEUE_SIZE 4096
#else
#define QUEUE_SIZE 8192
#endif

DEFINE_STATIC_BYTE_QUEUE(tx_queue, QUEUE_SIZE);
static int bytes_out;
static int max_packet;
static STATE state;

typedef enum {
  WAIT_TYPE,
  WAIT_ARGS,
  WAIT_VAR_ARGS
} RX_MESSAGE_STATE;

static INCOMING_MESSAGE rx_msg;
static int rx_buffer_cursor;
static int rx_message_remaining;
static RX_MESSAGE_STATE rx_message_state;
```

```
static inline BYTE OutgoingMessageLength(const OUTGOING_MESSAGE* msg) {
  return 1 + outgoing_arg_size[msg->type];
}

static inline BYTE IncomingVarArgSize(const INCOMING_MESSAGE* msg) {
  switch (msg->type) {
    case UART_DATA:
      return msg->args.uart_data.size + 1;

    case SPI_MASTER_REQUEST:
      if (msg->args.spi_master_request.data_size_neq_total) {
        return msg->args.spi_master_request.data_size
            + msg->args.spi_master_request.res_size_neq_total;
      } else {
        return msg->args.spi_master_request.total_size
            + msg->args.spi_master_request.res_size_neq_total;
      }

    case I2C_WRITE_READ:
      return msg->args.i2c_write_read.write_size;

    case SEQUENCER_CONFIGURE:
      return msg->args.sequencer_configure.size;

    case SEQUENCER_PUSH:
      return SequencerExpectedCueSize();

    case SEQUENCER_CONTROL:
      return msg->args.sequencer_control.cmd == SEQ_CMD_MANUAL_START
                                              ? SequencerExpectedCueSize()
                                              : 0;

    // BOOKMARK(add_feature): Add more cases here if incoming message has
variable args.
    default:
      return 0;
  }
}

void AppProtocolInit(CHANNEL_HANDLE h) {
  _prog_addressT p;
  bytes_out = 0;
  rx_buffer_cursor = 0;
  rx_message_remaining = 1;
  rx_message_state = WAIT_TYPE;
  ByteQueueClear(&tx_queue);
  max_packet = ConnectionGetMaxPacket(h);
  state = STATE_OPEN;

  OUTGOING_MESSAGE msg;
  msg.type = ESTABLISH_CONNECTION;
  msg.args.establish_connection.magic = IOIO_MAGIC;

  _init_prog_address(p, hardware_version);
  _memcpy_p2d16(msg.args.establish_connection.hw_impl_ver, p, 8);
  _init_prog_address(p, bootloader_version);
  _memcpy_p2d16(msg.args.establish_connection.bl_impl_ver, p, 8);

  memcpy(msg.args.establish_connection.fw_impl_ver, FW_IMPL_VER, 8);

  AppProtocolSendMessage(&msg);
```

```
}

void AppProtocolSendMessage(const OUTGOING_MESSAGE* msg) {
  if (state != STATE_OPEN) return;
  PRIORITY(1) {
    ByteQueuePushBuffer(&tx_queue, (const BYTE*) msg,
OutgoingMessageLength(msg));
  }
}

void AppProtocolSendMessageWithVarArg(const OUTGOING_MESSAGE* msg, const
void* data, int size) {
  if (state != STATE_OPEN) return;
  PRIORITY(1) {
    ByteQueuePushBuffer(&tx_queue, (const BYTE*) msg,
OutgoingMessageLength(msg));
    ByteQueuePushBuffer(&tx_queue, data, size);
  }
}

void AppProtocolSendMessageWithVarArgSplit(const OUTGOING_MESSAGE* msg,
                                           const void* data1, int size1,
                                           const void* data2, int size2) {
  if (state != STATE_OPEN) return;
  PRIORITY(1) {
    ByteQueuePushBuffer(&tx_queue, (const BYTE*) msg,
OutgoingMessageLength(msg));
    ByteQueuePushBuffer(&tx_queue, data1, size1);
    ByteQueuePushBuffer(&tx_queue, data2, size2);
  }
}

void AppProtocolTasks(CHANNEL_HANDLE h) {
  if (state == STATE_CLOSED) return;
  if (state == STATE_CLOSING && ByteQueueSize(&tx_queue) == 0) {
    log_printf("Finished flushing, closing the channel.");
    ConnectionCloseChannel(h);
    state = STATE_CLOSED;
    return;
  }
  UARTTasks();
  SPITasks();
  I2CTasks();
  ICSPTasks();
  SequencerTasks();
  if (ConnectionCanSend(h)) {
    const BYTE* data;
    if (bytes_out) {
      ByteQueuePull(&tx_queue, bytes_out);
      bytes_out = 0;
    }
    ByteQueuePeek(&tx_queue, &data, &bytes_out);
    if (bytes_out > 0) {
      if (bytes_out > max_packet) bytes_out = max_packet;
      ConnectionSend(h, data, bytes_out);
    }
  }
}

static void Echo() {
  AppProtocolSendMessage((const OUTGOING_MESSAGE*) &rx_msg);
}
```

```
static BOOL MessageDone() {
  // TODO: check pin capabilities
  switch (rx_msg.type) {
    case HARD_RESET:
      CHECK(rx_msg.args.hard_reset.magic == IOIO_MAGIC);
      HardReset();
      break;

    case SOFT_RESET:
      SoftReset();
      Echo();
      break;

    case SET_PIN_DIGITAL_OUT:
      CHECK(rx_msg.args.set_pin_digital_out.pin < NUM_PINS);
      SetPinDigitalOut(rx_msg.args.set_pin_digital_out.pin,
                       rx_msg.args.set_pin_digital_out.value,
                       rx_msg.args.set_pin_digital_out.open_drain);
      break;

    case SET_DIGITAL_OUT_LEVEL:
      CHECK(rx_msg.args.set_digital_out_level.pin < NUM_PINS);
      if(rx_msg.args.set_digital_out_level.value == 1) {
        SetDigitalOutLevel(rx_msg.args.set_digital_out_level.pin,
                       rx_msg.args.set_digital_out_level.value);
       }
       else
        SetDigitalOutLevel(rx_msg.args.set_digital_out_level.pin,
                   rx_msg.args.set_digital_out_level.value);

      break;

    case SET_PIN_DIGITAL_IN:
      CHECK(rx_msg.args.set_pin_digital_in.pin < NUM_PINS);
      CHECK(rx_msg.args.set_pin_digital_in.pull < 3);
      SetPinDigitalIn(rx_msg.args.set_pin_digital_in.pin,
rx_msg.args.set_pin_digital_in.pull);
      break;

    case SET_CHANGE_NOTIFY:
      CHECK(rx_msg.args.set_change_notify.pin < NUM_PINS);
      if (rx_msg.args.set_change_notify.cn) {
        Echo();
      }
      SetChangeNotify(rx_msg.args.set_change_notify.pin,
rx_msg.args.set_change_notify.cn);
      if (!rx_msg.args.set_change_notify.cn) {
        Echo();
      }
      break;

    case SET_PIN_PWM:
      CHECK(rx_msg.args.set_pin_pwm.pin < NUM_PINS);
      CHECK(rx_msg.args.set_pin_pwm.pwm_num < NUM_PWM_MODULES);
      SetPinPwm(rx_msg.args.set_pin_pwm.pin, rx_msg.args.set_pin_pwm.pwm_num,
                rx_msg.args.set_pin_pwm.enable);
      break;

    case SET_PWM_DUTY_CYCLE:
      CHECK(rx_msg.args.set_pwm_duty_cycle.pwm_num < NUM_PWM_MODULES);
      SetPwmDutyCycle(rx_msg.args.set_pwm_duty_cycle.pwm_num,
```

```
                    rx_msg.args.set_pwm_duty_cycle.dc,
                    rx_msg.args.set_pwm_duty_cycle.fraction);
      break;

    case SET_PWM_PERIOD:
      CHECK(rx_msg.args.set_pwm_period.pwm_num < NUM_PWM_MODULES);
      SetPwmPeriod(rx_msg.args.set_pwm_period.pwm_num,
                   rx_msg.args.set_pwm_period.period,
                   rx_msg.args.set_pwm_period.scale_l
                   | (rx_msg.args.set_pwm_period.scale_h) << 1);
      break;

    case SET_PIN_ANALOG_IN:
      CHECK(rx_msg.args.set_pin_analog_in.pin < NUM_PINS);
      SetPinAnalogIn(rx_msg.args.set_pin_analog_in.pin);
      break;

    case UART_DATA:
      CHECK(rx_msg.args.uart_data.uart_num < NUM_UART_MODULES);
      UARTTransmit(rx_msg.args.uart_data.uart_num,
                   rx_msg.args.uart_data.data,
                   rx_msg.args.uart_data.size + 1);
      break;

    case UART_CONFIG:
      CHECK(rx_msg.args.uart_config.uart_num < NUM_UART_MODULES);
      CHECK(rx_msg.args.uart_config.parity < 3);
      UARTConfig(rx_msg.args.uart_config.uart_num,
                 rx_msg.args.uart_config.rate,
                 rx_msg.args.uart_config.speed4x,
                 rx_msg.args.uart_config.two_stop_bits,
                 rx_msg.args.uart_config.parity);
      break;

    case SET_PIN_UART:
      CHECK(rx_msg.args.set_pin_uart.pin < NUM_PINS);
      CHECK(rx_msg.args.set_pin_uart.uart_num < NUM_UART_MODULES);
      SetPinUart(rx_msg.args.set_pin_uart.pin,
                 rx_msg.args.set_pin_uart.uart_num,
                 rx_msg.args.set_pin_uart.dir,
                 rx_msg.args.set_pin_uart.enable);
      break;

    case SPI_MASTER_REQUEST:
      CHECK(rx_msg.args.spi_master_request.spi_num < NUM_SPI_MODULES);
      CHECK(rx_msg.args.spi_master_request.ss_pin < NUM_PINS);
      {
        const BYTE total_size = rx_msg.args.spi_master_request.total_size +
1;
        const BYTE data_size =
rx_msg.args.spi_master_request.data_size_neq_total
            ? rx_msg.args.spi_master_request.data_size
            : total_size;
        const BYTE res_size =
rx_msg.args.spi_master_request.res_size_neq_total
            ? rx_msg.args.spi_master_request.vararg[
                rx_msg.args.spi_master_request.data_size_neq_total]
            : total_size;
        const BYTE* const data = &rx_msg.args.spi_master_request.vararg[
            rx_msg.args.spi_master_request.data_size_neq_total
            + rx_msg.args.spi_master_request.res_size_neq_total];
```

```
      SPITransmit(rx_msg.args.spi_master_request.spi_num,
                  rx_msg.args.spi_master_request.ss_pin,
                  data,
                  data_size,
                  total_size,
                  total_size - res_size);
    }
    break;

  case SPI_CONFIGURE_MASTER:
    CHECK(rx_msg.args.spi_configure_master.spi_num < NUM_SPI_MODULES);
    SPIConfigMaster(rx_msg.args.spi_configure_master.spi_num,
                    rx_msg.args.spi_configure_master.scale,
                    rx_msg.args.spi_configure_master.div,
                    rx_msg.args.spi_configure_master.smp_end,
                    rx_msg.args.spi_configure_master.clk_edge,
                    rx_msg.args.spi_configure_master.clk_pol);
    break;

  case SET_PIN_SPI:
    CHECK(rx_msg.args.set_pin_spi.mode < 3);
    CHECK((!rx_msg.args.set_pin_spi.enable
          && rx_msg.args.set_pin_spi.mode == 1)
          || rx_msg.args.set_pin_spi.pin < NUM_PINS);
    CHECK((!rx_msg.args.set_pin_spi.enable
          && rx_msg.args.set_pin_spi.mode != 1)
          || rx_msg.args.set_pin_spi.spi_num < NUM_SPI_MODULES);
    SetPinSpi(rx_msg.args.set_pin_spi.pin,
              rx_msg.args.set_pin_spi.spi_num,
              rx_msg.args.set_pin_spi.mode,
              rx_msg.args.set_pin_spi.enable);
    break;

  case I2C_CONFIGURE_MASTER:
    CHECK(rx_msg.args.i2c_configure_master.i2c_num < NUM_I2C_MODULES);
    I2CConfigMaster(rx_msg.args.i2c_configure_master.i2c_num,
                    rx_msg.args.i2c_configure_master.rate,
                    rx_msg.args.i2c_configure_master.smbus_levels);
    break;

  case I2C_WRITE_READ:
    CHECK(rx_msg.args.i2c_write_read.i2c_num < NUM_I2C_MODULES);
    {
      unsigned int addr;
      if (rx_msg.args.i2c_write_read.ten_bit_addr) {
        addr = rx_msg.args.i2c_write_read.addr_lsb;
        addr = addr << 8
                | ((rx_msg.args.i2c_write_read.addr_msb << 1)
                  | 0b11110000);
      } else {
        CHECK(rx_msg.args.i2c_write_read.addr_msb == 0
              && rx_msg.args.i2c_write_read.addr_lsb >> 7 == 0
              && rx_msg.args.i2c_write_read.addr_lsb >> 2 != 0b0011110);
        addr = rx_msg.args.i2c_write_read.addr_lsb << 1;
      }
      I2CWriteRead(rx_msg.args.i2c_write_read.i2c_num,
                   addr,
                   rx_msg.args.i2c_write_read.data,
                   rx_msg.args.i2c_write_read.write_size,
                   rx_msg.args.i2c_write_read.read_size);
    }
    break;
```

```
    case SET_ANALOG_IN_SAMPLING:
      CHECK(rx_msg.args.set_analog_pin_sampling.pin < NUM_PINS);
      ADCSetScan(rx_msg.args.set_analog_pin_sampling.pin,
                 rx_msg.args.set_analog_pin_sampling.enable);
      break;

    case SET_ANALOG_OFFLINE:
      ADCtype=1;
      ADCOfflineSetScan();
      break;

    case SET_RETURN_ANALOG_OFFLINE_STATUS:
      ADCtype=0;
      ReportADCResults();
      break;

    case CHECK_INTERFACE:
      CheckInterface(rx_msg.args.check_interface.interface_id);
      break;

    case ICSP_SIX:
      ICSPSix(rx_msg.args.icsp_six.inst);
      break;

    case ICSP_REGOUT:
      ICSPRegout();
      break;

    case ICSP_PROG_ENTER:
      ICSPEnter();
      break;

    case ICSP_PROG_EXIT:
      ICSPExit();
      break;

    case ICSP_CONFIG:
      if (rx_msg.args.icsp_config.enable) {
        Echo();
      }
      ICSPConfigure(rx_msg.args.icsp_config.enable);
      if (!rx_msg.args.icsp_config.enable) {
        Echo();
      }
      break;

    case INCAP_CONFIG:
      CHECK(rx_msg.args.incap_config.incap_num < NUM_INCAP_MODULES);
      CHECK(!rx_msg.args.incap_config.double_prec
            || 0 == (rx_msg.args.incap_config.incap_num & 0x01));
      CHECK(rx_msg.args.incap_config.mode < 6);
      CHECK(rx_msg.args.incap_config.clock < 4);
      InCapConfig(rx_msg.args.incap_config.incap_num,
                  rx_msg.args.incap_config.double_prec,
                  rx_msg.args.incap_config.mode,
                  rx_msg.args.incap_config.clock);
      break;

    case SET_PIN_INCAP:
      CHECK(rx_msg.args.set_pin_incap.incap_num < NUM_INCAP_MODULES);
      CHECK(!rx_msg.args.set_pin_incap.enable
```

```
            || rx_msg.args.set_pin_incap.pin < NUM_PINS);
      SetPinInCap(rx_msg.args.set_pin_incap.pin,
                  rx_msg.args.set_pin_incap.incap_num,
                  rx_msg.args.set_pin_incap.enable);
      break;

    case SOFT_CLOSE:
      log_printf("Soft close requested");
      Echo();
      state = STATE_CLOSING;
      break;

    case SET_PIN_CAPSENSE:
      CHECK(rx_msg.args.set_pin_capsense.pin < NUM_PINS);
      SetPinCapSense(rx_msg.args.set_pin_capsense.pin);
      break;

    case SET_CAPSENSE_SAMPLING:
      CHECK(rx_msg.args.set_capsense_sampling.pin < NUM_PINS);
      ADCSetCapSense(rx_msg.args.set_capsense_sampling.pin,
                     rx_msg.args.set_capsense_sampling.enable);
      break;

    case SEQUENCER_CONFIGURE:
      if (rx_msg.args.sequencer_configure.size) {
        return SequencerOpen(rx_msg.args.sequencer_configure.config,
                             rx_msg.args.sequencer_configure.size);
      } else {
        return SequencerClose();
      }

    case SEQUENCER_PUSH:
      return SequencerPush(rx_msg.args.sequencer_push.cue,
                           rx_msg.args.sequencer_push.time);

    case SEQUENCER_CONTROL:
      return SequencerCommand((SEQ_CMD) rx_msg.args.sequencer_control.cmd,
                              rx_msg.args.sequencer_control.extra);

    case SYNC:
      Echo();
      break;

    // BOOKMARK(add_feature): Add incoming message handling to switch clause.
    // Call Echo() if the message is to be echoed back.

    default:
      return FALSE;
  }
  return TRUE;
}

BOOL AppProtocolHandleIncoming(const BYTE* data, UINT32 data_len) {
  assert(data);
  if (state != STATE_OPEN) {
    log_printf("Shouldn't get data after close!");
    return FALSE;
  }

  while (data_len > 0) {
    // copy a chunk of data to rx_msg
    if (data_len >= rx_message_remaining) {
```

```
      memcpy(((BYTE *) &rx_msg) + rx_buffer_cursor, data,
rx_message_remaining);
      data += rx_message_remaining;
      data_len -= rx_message_remaining;
      rx_buffer_cursor += rx_message_remaining;
      rx_message_remaining = 0;
    } else {
      memcpy(((BYTE *) &rx_msg) + rx_buffer_cursor, data, data_len);
      rx_buffer_cursor += data_len;
      rx_message_remaining -= data_len;
      data_len = 0;
    }

    // change state
    if (rx_message_remaining == 0) {
      switch (rx_message_state) {
        case WAIT_TYPE:
          rx_message_state = WAIT_ARGS;
          rx_message_remaining = incoming_arg_size[rx_msg.type];
          if (rx_message_remaining) break;
          // fall-through on purpose

        case WAIT_ARGS:
          rx_message_state = WAIT_VAR_ARGS;
          rx_message_remaining = IncomingVarArgSize(&rx_msg);
          if (rx_message_remaining) break;
          // fall-through on purpose

        case WAIT_VAR_ARGS:
          rx_message_state = WAIT_TYPE;
          rx_message_remaining = 1;
          rx_buffer_cursor = 0;
          if (!MessageDone()) return FALSE;
          break;
      }
    }
  }
  return TRUE;
}
```

features.c

```
#include "features.h"

#include <string.h>

#include "Compiler.h"
#include "pins.h"
#include "logging.h"
#include "protocol.h"
#include "adc.h"
#include "pwm.h"
#include "uart.h"
#include "spi.h"
#include "i2c.h"
#include "timers.h"
#include "sequencer.h"
#include "pp_util.h"
#include "incap.h"
```

```
#include "sync.h"


extern int ADCtype;

////////////////////////////////////////////////////////////////////////////////
// Pin modes
////////////////////////////////////////////////////////////////////////////////

static void PinsInit() {
  int i;
  _CNIE = 0;
  // reset pin states
  SetPinDigitalOut(0, 1, 1);  // LED pin: output, open-drain, high (off)

  for (i = 1; i < NUM_PINS; ++i) {
      // Pin 36: ADCOffline
      if((ADCtype==1 && i!=36) || ADCtype==0)
        SetPinDigitalIn(i, 0);    // all other pins: input, no-pull
  }
  // clear and enable global CN interrupts
  _CNIF = 0;
  _CNIE = 1;
  _CNIP = 1;  // CN interrupt priority is 1 so it can write an outgoing
message
}

void SetPinDigitalOut(int pin, int value, int open_drain) {
  log_printf("SetPinDigitalOut(%d, %d, %d)", pin, value, open_drain);
  SAVE_PIN_FOR_LOG(pin);
  // Protect from the user trying to use push-pull, later pulling low using
the
  // bootloader pin.
  if (pin == 0) open_drain = 1;
  PinSetAnsel(pin, 0);
  PinSetRpor(pin, 0);
  PinSetCnen(pin, 0);
  PinSetCnpu(pin, 0);
  PinSetCnpd(pin, 0);
  PinSetLat(pin, value);
  PinSetOdc(pin, open_drain);
  PinSetTris(pin, 0);
}

void SetPinDigitalIn(int pin, int pull) {
  log_printf("SetPinDigitalIn(%d, %d)", pin, pull);
  SAVE_PIN_FOR_LOG(pin);
  PinSetAnsel(pin, 0);
  PinSetRpor(pin, 0);
  PinSetCnen(pin, 0);
  switch (pull) {
    case 1:
      PinSetCnpd(pin, 0);
      PinSetCnpu(pin, 1);
      break;

    case 2:
      PinSetCnpu(pin, 0);
      PinSetCnpd(pin, 1);
      break;

    default:
```

```
      PinSetCnpu(pin, 0);
      PinSetCnpd(pin, 0);
  }
  PinSetTris(pin, 1);
}

void SetPinPwm(int pin, int pwm_num, int enable) {
  log_printf("SetPinPwm(%d, %d)", pin, pwm_num);
  SAVE_PIN_FOR_LOG(pin);
  PinSetRpor(pin, enable ? (pwm_num == 8 ? 35 : 18 + pwm_num) : 0);
}

void SetPinUart(int pin, int uart_num, int dir, int enable) {
  log_printf("SetPinUart(%d, %d, %d, %d)", pin, uart_num, dir, enable);
  SAVE_PIN_FOR_LOG(pin);
  SAVE_UART_FOR_LOG(uart_num);
  if (dir) {
    // TX
    const BYTE rp[] = { 3, 5, 28, 30 };
    PinSetRpor(pin, enable ? rp[uart_num] : 0);
  } else {
    // RX
    int rpin = enable ? PinToRpin(pin) : 0x3F;
    switch (uart_num) {
      case 0:
        _U1RXR = rpin;
        break;

      case 1:
        _U2RXR = rpin;
        break;

      case 2:
        _U3RXR = rpin;
        break;

      case 3:
        _U4RXR = rpin;
        break;
    }
  }
}

void SetPinInCap(int pin, int incap_num, int enable) {
  log_printf("SetPinInCap(%d, %d, %d)", pin, incap_num, enable);
    int rpin = enable ? PinToRpin(pin) : 0x3F;

    switch (incap_num) {
#define CASE(num, unused)     \
      case num - 1:           \
        _IC##num##R = rpin;   \
        break;
      REPEAT_1B(CASE, NUM_INCAP_MODULES);
    }
}

void SetPinAnalogIn(int pin) {
  log_printf("SetPinAnalogIn(%d)", pin);
  SAVE_PIN_FOR_LOG(pin);
  PinSetRpor(pin, 0);
  PinSetCnen(pin, 0);
  PinSetCnpu(pin, 0);
```

```
  PinSetCnpd(pin, 0);
  PinSetAnsel(pin, 1);
  PinSetTris(pin, 1);
}

void SetPinCapSense(int pin) {
  log_printf("SetPinCapSense(%d)", pin);
  SAVE_PIN_FOR_LOG(pin);

  // In cap-sense mode, the pin is configured for analog input, but initially
  // pulling low in order to discharge, the circuit.
  PinSetRpor(pin, 0);
  PinSetCnen(pin, 0);
  PinSetCnpu(pin, 0);
  PinSetCnpd(pin, 0);
  PinSetAnsel(pin, 1);
  PinSetLat(pin, 0);
  PinSetTris(pin, 1);
}

void SetPinSpi(int pin, int spi_num, int mode, int enable) {
  log_printf("SetPinSpi(%d, %d, %d, %d)", pin, spi_num, mode, enable);
  SAVE_PIN_FOR_LOG(pin);
  switch (mode) {
    case 0:  // data out
      {
        const BYTE rp[] = { 7, 10, 32 };
        PinSetRpor(pin, enable ? rp[spi_num] : 0);
      }
      break;

      case 1:  // data in
      {
        int rpin = enable ? PinToRpin(pin) : 0x3F;
        switch (spi_num) {
          case 0:
            _SDI1R = rpin;
            break;

          case 1:
            _SDI2R = rpin;
            break;

          case 2:
            _SDI3R = rpin;
            break;
        }
      }
      break;

      case 2:  // clk out
      {
        const BYTE rp[] = { 8, 11, 33 };
        PinSetRpor(pin, enable ? rp[spi_num] : 0);
      }
      break;
  }
}

////////////////////////////////////////////////////////////////////////////
// Reset
////////////////////////////////////////////////////////////////////////////
```

```
void HardReset() {
  log_printf("HardReset()");
  log_printf("Rebooting...");
  Reset();
}

void SoftReset() {
  PRIORITY(7) {
    log_printf("SoftReset()");
    TimersInit();
    PinsInit();
    PWMInit();
    if(ADCtype==0)
        ADCInit();
    UARTInit();
    SPIInit();
    I2CInit();
    InCapInit();
    SequencerInit();

    // TODO: reset all peripherals!
  }
}

void CheckInterface(BYTE interface_id[8]) {
  OUTGOING_MESSAGE msg;
  msg.type = CHECK_INTERFACE_RESPONSE;
  msg.args.check_interface_response.supported
      = (memcmp(interface_id, PROTOCOL_IID_IOIO0005, 8) == 0)
        || (memcmp(interface_id, PROTOCOL_IID_IOIO0004, 8) == 0)
        || (memcmp(interface_id, PROTOCOL_IID_IOIO0003, 8) == 0)
        || (memcmp(interface_id, PROTOCOL_IID_IOIO0002, 8) == 0)
        || (memcmp(interface_id, PROTOCOL_IID_IOIO0001, 8) == 0);
  AppProtocolSendMessage(&msg);
}

// BOOKMARK(add_feature): Add feature implementation.
```

2. Librería

IOIOProtocol.java

```
package ioio.lib.impl;

import ioio.lib.api.DigitalInput;
import ioio.lib.api.DigitalOutput;
import ioio.lib.api.SpiMaster;
import ioio.lib.api.TwiMaster.Rate;
import ioio.lib.api.Uart;
import ioio.lib.spi.Log;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Set;

class IOIOProtocol {
      static final int HARD_RESET                          = 0x00;
      static final int ESTABLISH_CONNECTION                = 0x00;
      static final int SOFT_RESET                          = 0x01;
      static final int CHECK_INTERFACE                     = 0x02;
      static final int CHECK_INTERFACE_RESPONSE            = 0x02;
      static final int SET_PIN_DIGITAL_OUT                 = 0x03;
      static final int SET_DIGITAL_OUT_LEVEL               = 0x04;
      static final int REPORT_DIGITAL_IN_STATUS            = 0x04;
      static final int SET_PIN_DIGITAL_IN                  = 0x05;
      static final int REPORT_PERIODIC_DIGITAL_IN_STATUS   = 0x05;
      static final int SET_CHANGE_NOTIFY                   = 0x06;
      static final int REGISTER_PERIODIC_DIGITAL_SAMPLING  = 0x07;
      static final int SET_PIN_PWM                         = 0x08;
      static final int SET_PWM_DUTY_CYCLE                  = 0x09;
      static final int SET_PWM_PERIOD                      = 0x0A;
      static final int SET_PIN_ANALOG_IN                   = 0x0B;
      static final int REPORT_ANALOG_IN_STATUS             = 0x0B;
      static final int SET_ANALOG_IN_SAMPLING              = 0x0C;
      static final int REPORT_ANALOG_IN_FORMAT             = 0x0C;
      static final int UART_CONFIG                         = 0x0D;
      static final int UART_STATUS                         = 0x0D;
      static final int UART_DATA                           = 0x0E;
      static final int SET_PIN_UART                        = 0x0F;
      static final int UART_REPORT_TX_STATUS               = 0x0F;
      static final int SPI_CONFIGURE_MASTER                = 0x10;
      static final int SPI_STATUS                          = 0x10;
      static final int SPI_MASTER_REQUEST                  = 0x11;
      static final int SPI_DATA                            = 0x11;
      static final int SET_PIN_SPI                         = 0x12;
      static final int SPI_REPORT_TX_STATUS                = 0x12;
      static final int I2C_CONFIGURE_MASTER                = 0x13;
      static final int I2C_STATUS                          = 0x13;
      static final int I2C_WRITE_READ                      = 0x14;
      static final int I2C_RESULT                          = 0x14;
      static final int I2C_REPORT_TX_STATUS                = 0x15;
      static final int ICSP_SIX                            = 0x16;
      static final int ICSP_REPORT_RX_STATUS               = 0x16;
      static final int ICSP_REGOUT                         = 0x17;
      static final int ICSP_RESULT                         = 0x17;
```

```
	static final int ICSP_PROG_ENTER                    = 0x18;
	static final int ICSP_PROG_EXIT                     = 0x19;
	static final int ICSP_CONFIG                        = 0x1A;
	static final int INCAP_CONFIGURE                    = 0x1B;
	static final int INCAP_STATUS                       = 0x1B;
	static final int SET_PIN_INCAP                      = 0x1C;
	static final int INCAP_REPORT                       = 0x1C;
	static final int SOFT_CLOSE                         = 0x1D;
	static final int SET_PIN_CAPSENSE                   = 0x1E;
	static final int CAPSENSE_REPORT                    = 0x1E;
	static final int SET_CAPSENSE_SAMPLING              = 0x1F;
	static final int SEQUENCER_CONFIGURE                = 0x20;
	static final int SEQUENCER_EVENT                    = 0x20;
	static final int SEQUENCER_PUSH                     = 0x21;
	static final int SEQUENCER_CONTROL                  = 0x22;
	static final int SYNC                               = 0x23;
	static final int SET_ANALOG_OFFLINE                 = 0x24;
	static final int SET_RETURN_ANALOG_OFFLINE_STATUS   = 0x25;
	static final int REPORT_ANALOG_OFFLINE_STATUS       = 0x26;

	static final int[] SCALE_DIV = new int[] {
		0x1F,  // 31.25
		0x1E,  // 35.714
		0x1D,  // 41.667
		0x1C,  // 50
		0x1B,  // 62.5
		0x1A,  // 83.333
		0x17,  // 125
		0x16,  // 142.857
		0x15,  // 166.667
		0x14,  // 200
		0x13,  // 250
		0x12,  // 333.333
		0x0F,  // 500
		0x0E,  // 571.429
		0x0D,  // 666.667
		0x0C,  // 800
		0x0B,  // 1000
		0x0A,  // 1333.333
		0x07,  // 2000
		0x06,  // 2285.714
		0x05,  // 2666.667
		0x04,  // 3200
		0x03,  // 4000
		0x02,  // 5333.333
		0x01   // 8000
	};

	private static final String TAG = "IOIOProtocol";
	private static final int AnalogMsgLength = 125;

	enum PwmScale {
		SCALE_1X(1, 0), SCALE_8X(8, 3), SCALE_64X(64, 2), SCALE_256X(256,
1);

		public final int scale;
		private final int encoding;

		PwmScale(int scale, int encoding) {
			this.scale = scale;
			this.encoding = encoding;
		}
```

```
	}

	enum SequencerEvent {
		PAUSED, STALLED, OPENED, NEXT_CUE, STOPPED, CLOSED
	}

	static class ProtocolError extends Exception {
		private static final long serialVersionUID = -
6973476719285599189L;

		public ProtocolError() {
			super();
		}

		public ProtocolError(String msg) {
			super(msg);
		}

		public ProtocolError(Exception e) {
			super(e);
		}
	}

	private int batchCounter_ = 0;

	private void writeByte(int b) throws IOException {
		assert (b >= 0 && b < 256);
		Log.v(TAG, "sending: 0x" + Integer.toHexString(b));
		out_.write(b);
	}

	private void writeBytes(byte[] buf, int offset, int size) throws
IOException {
		while (size-- > 0) {
			writeByte(((int) buf[offset++]) & 0xFF);
		}
	}

	public synchronized void beginBatch() {
		++batchCounter_;
	}

	public synchronized void endBatch() throws IOException {
		if (--batchCounter_ == 0) {
			out_.flush();
		}
	}

	private void writeTwoBytes(int i) throws IOException {
		writeByte(i & 0xFF);
		writeByte(i >> 8);
	}

	private void writeThreeBytes(int i) throws IOException {
		writeByte(i & 0xFF);
		writeByte((i >> 8) & 0xFF);
		writeByte((i >> 16) & 0xFF);
	}

	public synchronized void sync() throws IOException {
		beginBatch();
		writeByte(SYNC);
```

```
            endBatch();
      }

      synchronized public void hardReset() throws IOException {
            beginBatch();
            writeByte(HARD_RESET);
            writeByte('I');
            writeByte('O');
            writeByte('I');
            writeByte('O');
            endBatch();
      }

      synchronized public void softReset() throws IOException {
            beginBatch();
            writeByte(SOFT_RESET);
            endBatch();
      }

      synchronized public void softClose() throws IOException {
            beginBatch();
            writeByte(SOFT_CLOSE);
            endBatch();
      }

      synchronized public void checkInterface(byte[] interfaceId) throws
IOException {
            if (interfaceId.length != 8) {
                  throw new IllegalArgumentException("interface ID must be
exactly 8 bytes long");
            }
            beginBatch();
            writeByte(CHECK_INTERFACE);
            for (int i = 0; i < 8; ++i) {
                  writeByte(interfaceId[i]);
            }
            endBatch();
      }

      synchronized public void setDigitalOutLevel(int pin, boolean level)
throws IOException {
            beginBatch();
            writeByte(SET_DIGITAL_OUT_LEVEL);
            writeByte(pin << 2 | (level ? 1 : 0));
            endBatch();
      }

      synchronized public void setPinPwm(int pin, int pwmNum, boolean enable)
throws IOException {
            beginBatch();
            writeByte(SET_PIN_PWM);
            writeByte(pin & 0x3F);
            writeByte((enable ? 0x80 : 0x00) | (pwmNum & 0x0F));
            endBatch();
      }

      synchronized public void setPwmDutyCycle(int pwmNum, int dutyCycle, int
fraction)
                  throws IOException {
            beginBatch();
            writeByte(SET_PWM_DUTY_CYCLE);
            writeByte(pwmNum << 2 | fraction);
```

```
            writeTwoBytes(dutyCycle);
            endBatch();
      }

      synchronized public void setPwmPeriod(int pwmNum, int period, PwmScale
scale)
                  throws IOException {
            beginBatch();
            writeByte(SET_PWM_PERIOD);
            writeByte(((scale.encoding & 0x02) << 6) | (pwmNum << 1) |
(scale.encoding & 0x01));
            writeTwoBytes(period);
            endBatch();
      }

      synchronized public void setPinIncap(int pin, int incapNum, boolean
enable) throws IOException {
            beginBatch();
            writeByte(SET_PIN_INCAP);
            writeByte(pin);
            writeByte(incapNum | (enable ? 0x80 : 0x00));
            endBatch();
      }

      synchronized public void incapClose(int incapNum, boolean double_prec)
throws IOException {
            beginBatch();
            writeByte(INCAP_CONFIGURE);
            writeByte(incapNum);
            writeByte(double_prec ? 0x80 : 0x00);
            endBatch();
      }

      synchronized public void incapConfigure(int incapNum, boolean
double_prec, int mode, int clock)
                  throws IOException {
            beginBatch();
            writeByte(INCAP_CONFIGURE);
            writeByte(incapNum);
            writeByte((double_prec ? 0x80 : 0x00) | (mode << 3) | clock);
            endBatch();
      }


      synchronized public void i2cWriteRead(int i2cNum, boolean tenBitAddr,
int address,
                  int writeSize, int readSize, byte[] writeData) throws
IOException {
            beginBatch();
            writeByte(I2C_WRITE_READ);
            writeByte(((address >> 8) << 6) | (tenBitAddr ? 0x20 : 0x00) |
i2cNum);
            writeByte(address & 0xFF);
            writeByte(writeSize);
            writeByte(readSize);
            for (int i = 0; i < writeSize; ++i) {
                  writeByte(((int) writeData[i]) & 0xFF);
            }
            endBatch();
      }

      synchronized public void setPinDigitalOut(int pin, boolean value,
```

```
DigitalOutput.Spec.Mode mode)
                  throws IOException {
            beginBatch();
            writeByte(SET_PIN_DIGITAL_OUT);
            writeByte((pin << 2) | (mode == DigitalOutput.Spec.Mode.OPEN_DRAIN
? 0x01 : 0x00)
                        | (value ? 0x02 : 0x00));
            endBatch();
      }

      synchronized public void setPinDigitalIn(int pin,
DigitalInput.Spec.Mode mode)
                  throws IOException {
            int pull = 0;
            if (mode == DigitalInput.Spec.Mode.PULL_UP) {
                  pull = 1;
            } else if (mode == DigitalInput.Spec.Mode.PULL_DOWN) {
                  pull = 2;
            }
            beginBatch();
            writeByte(SET_PIN_DIGITAL_IN);
            writeByte((pin << 2) | pull);
            endBatch();
      }

      synchronized public void setChangeNotify(int pin, boolean changeNotify)
throws IOException {
            beginBatch();
            writeByte(SET_CHANGE_NOTIFY);
            writeByte((pin << 2) | (changeNotify ? 0x01 : 0x00));
            endBatch();
      }

      synchronized public void registerPeriodicDigitalSampling(int pin, int
freqScale)
                  throws IOException {
            // TODO: implement
      }

      synchronized public void setPinAnalogIn(int pin) throws IOException {
            beginBatch();
            writeByte(SET_PIN_ANALOG_IN);
            writeByte(pin);
            endBatch();
      }

      synchronized public void setAnalogInSampling(int pin, boolean enable)
throws IOException {
            beginBatch();
            writeByte(SET_ANALOG_IN_SAMPLING);
            writeByte((enable ? 0x80 : 0x00) | (pin & 0x3F));
            endBatch();
      }

      synchronized public void uartData(int uartNum, int numBytes, byte
data[]) throws IOException {
            if (numBytes > 64) {
                  throw new IllegalArgumentException(
                              "A maximum of 64 bytes can be sent in one
uartData message. Got: " + numBytes);
            }
            beginBatch();
```

```
            writeByte(UART_DATA);
            writeByte((numBytes - 1) | uartNum << 6);
            for (int i = 0; i < numBytes; ++i) {
                  writeByte(((int) data[i]) & 0xFF);
            }
            endBatch();
      }

      synchronized public void uartConfigure(int uartNum, int rate, boolean
speed4x,
                  Uart.StopBits stopbits, Uart.Parity parity) throws
IOException {
            int parbits = parity == Uart.Parity.EVEN ? 1 : (parity ==
Uart.Parity.ODD ? 2 : 0);
            beginBatch();
            writeByte(UART_CONFIG);
            writeByte((uartNum << 6) | (speed4x ? 0x08 : 0x00)
                        | (stopbits == Uart.StopBits.TWO ? 0x04 : 0x00) |
parbits);
            writeTwoBytes(rate);
            endBatch();
      }

      synchronized public void uartClose(int uartNum) throws IOException {
            beginBatch();
            writeByte(UART_CONFIG);
            writeByte(uartNum << 6);
            writeTwoBytes(0);
            endBatch();
      }

      synchronized public void setPinUart(int pin, int uartNum, boolean tx,
boolean enable)
                  throws IOException {
            beginBatch();
            writeByte(SET_PIN_UART);
            writeByte(pin);
            writeByte((enable ? 0x80 : 0x00) | (tx ? 0x40 : 0x00) | uartNum);
            endBatch();
      }

      synchronized public void spiConfigureMaster(int spiNum,
SpiMaster.Config config)
                  throws IOException {
            beginBatch();
            writeByte(SPI_CONFIGURE_MASTER);
            writeByte((spiNum << 5) | SCALE_DIV[config.rate.ordinal()]);
            writeByte((config.sampleOnTrailing ? 0x00 : 0x02) |
(config.invertClk ? 0x01 : 0x00));
            endBatch();
      }

      synchronized public void spiClose(int spiNum) throws IOException {
            beginBatch();
            writeByte(SPI_CONFIGURE_MASTER);
            writeByte(spiNum << 5);
            writeByte(0x00);
            endBatch();
      }

      synchronized public void setPinSpi(int pin, int mode, boolean enable,
int spiNum)
```

```
                  throws IOException {
            beginBatch();
            writeByte(SET_PIN_SPI);
            writeByte(pin);
            writeByte((1 << 4) | (mode << 2) | spiNum);
            endBatch();
      }

      synchronized public void spiMasterRequest(int spiNum, int ssPin, byte
data[], int dataBytes,
                  int totalBytes, int responseBytes) throws IOException {
            final boolean dataNeqTotal = (dataBytes != totalBytes);
            final boolean resNeqTotal = (responseBytes != totalBytes);
            beginBatch();
            writeByte(SPI_MASTER_REQUEST);
            writeByte((spiNum << 6) | ssPin);
            writeByte((dataNeqTotal ? 0x80 : 0x00) | (resNeqTotal ? 0x40 :
0x00) | totalBytes - 1);
            if (dataNeqTotal) {
                  writeByte(dataBytes);
            }
            if (resNeqTotal) {
                  writeByte(responseBytes);
            }
            for (int i = 0; i < dataBytes; ++i) {
                  writeByte(((int) data[i]) & 0xFF);
            }
            endBatch();
      }

      synchronized public void i2cConfigureMaster(int i2cNum, Rate rate,
boolean smbusLevels)
                  throws IOException {
            int rateBits = (rate == Rate.RATE_1MHz ? 3 : (rate ==
Rate.RATE_400KHz ? 2 : 1));
            beginBatch();
            writeByte(I2C_CONFIGURE_MASTER);
            writeByte((smbusLevels ? 0x80 : 0) | (rateBits << 5) | i2cNum);
            endBatch();
      }

      synchronized public void i2cClose(int i2cNum) throws IOException {
            beginBatch();
            writeByte(I2C_CONFIGURE_MASTER);
            writeByte(i2cNum);
            endBatch();
      }

      synchronized public void icspOpen() throws IOException {
            beginBatch();
            writeByte(ICSP_CONFIG);
            writeByte(0x01);
            endBatch();
      }

      synchronized public void icspClose() throws IOException {
            beginBatch();
            writeByte(ICSP_CONFIG);
            writeByte(0x00);
            endBatch();
      }
```

```
	synchronized public void icspEnter() throws IOException {
		beginBatch();
		writeByte(ICSP_PROG_ENTER);
		endBatch();
	}

	synchronized public void icspExit() throws IOException {
		beginBatch();
		writeByte(ICSP_PROG_EXIT);
		endBatch();
	}

	synchronized public void icspSix(int instruction) throws IOException {
		beginBatch();
		writeByte(ICSP_SIX);
		writeThreeBytes(instruction);
		endBatch();
	}

	synchronized public void icspRegout() throws IOException {
		beginBatch();
		writeByte(ICSP_REGOUT);
		endBatch();
	}

	synchronized public void setPinCapSense(int pinNum) throws IOException
{
		beginBatch();
		writeByte(SET_PIN_CAPSENSE);
		writeByte(pinNum & 0x3F);
		endBatch();
	}

	synchronized public void setCapSenseSampling(int pinNum, boolean
enable) throws IOException {
		beginBatch();
		writeByte(SET_CAPSENSE_SAMPLING);
		writeByte((pinNum & 0x3F) | (enable ? 0x80 : 0x00));
		endBatch();
	}

	synchronized public void sequencerOpen(byte[] config, int size) throws
IOException {
		assert config != null;
		assert size >= 0 && size <= 68;

		beginBatch();
		writeByte(SEQUENCER_CONFIGURE);
		writeByte(size);
		writeBytes(config, 0, size);
		endBatch();
	}

	synchronized public void sequencerClose() throws IOException {
		beginBatch();
		writeByte(SEQUENCER_CONFIGURE);
		writeByte(0);
		endBatch();
	}

	synchronized public void sequencerPush(int duration, byte[] cue, int
size) throws IOException {
```

```
            assert cue != null;
            assert size >= 0 && size <= 68;
            assert duration < (1 << 16);

            beginBatch();
            writeByte(SEQUENCER_PUSH);
            writeTwoBytes(duration);
            writeBytes(cue, 0, size);
            endBatch();
      }

      synchronized public void sequencerStop() throws IOException {
            beginBatch();
            writeByte(SEQUENCER_CONTROL);
            writeByte(0);
            endBatch();
      }

      synchronized public void sequencerStart() throws IOException {
            beginBatch();
            writeByte(SEQUENCER_CONTROL);
            writeByte(1);
            endBatch();
      }

      synchronized public void sequencerPause() throws IOException {
            beginBatch();
            writeByte(SEQUENCER_CONTROL);
            writeByte(2);
            endBatch();
      }

      synchronized public void sequencerManualStart(byte[] cue, int size)
throws IOException {
            beginBatch();
            writeByte(SEQUENCER_CONTROL);
            writeByte(3);
            writeBytes(cue, 0, size);
            endBatch();
      }

      synchronized public void sequencerManualStop() throws IOException {
            beginBatch();
            writeByte(SEQUENCER_CONTROL);
            writeByte(4);
            endBatch();
      }
      /**
       * MODIFICACIONES REALIZADAS

      public int getADCResult() {
            return ADCResult;
      }
      */

      synchronized public void setAnalogOffline() throws IOException {
            beginBatch();
            writeByte(SET_ANALOG_OFFLINE);
            endBatch();
      }

      synchronized public void setReturnAnalogOfflineStatus() throws
```

```
IOException {
            beginBatch();
            writeByte(SET_RETURN_ANALOG_OFFLINE_STATUS);
            endBatch();
      }

      public interface IncomingHandler {
            public void handleEstablishConnection(byte[] hardwareId, byte[]
bootloaderId,
                        byte[] firmwareId);

            public void handleConnectionLost();

            public void handleSoftReset();

            public void handleCheckInterfaceResponse(boolean supported);

            public void handleSetChangeNotify(int pin, boolean changeNotify);

            public void handleReportDigitalInStatus(int pin, boolean level);

            public void handleRegisterPeriodicDigitalSampling(int pin, int
freqScale);

            public void handleReportPeriodicDigitalInStatus(int frameNum,
boolean values[]);

            public void handleAnalogPinStatus(int pin, boolean open);

            public void handleReportAnalogInStatus(List<Integer> pins,
List<Integer> values);

            public void handleUartOpen(int uartNum);

            public void handleUartClose(int uartNum);

            public void handleUartData(int uartNum, int numBytes, byte
data[]);

            public void handleUartReportTxStatus(int uartNum, int
bytesRemaining);

            public void handleSpiOpen(int spiNum);

            public void handleSpiClose(int spiNum);

            public void handleSpiData(int spiNum, int ssPin, byte data[], int
dataBytes);

            public void handleSpiReportTxStatus(int spiNum, int
bytesRemaining);

            public void handleI2cOpen(int i2cNum);

            public void handleI2cClose(int i2cNum);

            public void handleI2cResult(int i2cNum, int size, byte[] data);

            public void handleI2cReportTxStatus(int spiNum, int
bytesRemaining);

            void handleIcspOpen();
```

```
            void handleIcspClose();

            void handleIcspReportRxStatus(int bytesRemaining);

            void handleIcspResult(int size, byte[] data);

            public void handleIncapReport(int incapNum, int size, byte[]
data);

            public void handleIncapClose(int incapNum);

            public void handleIncapOpen(int incapNum);

            public void handleCapSenseReport(int pinNum, int value);

            public void handleSetCapSenseSampling(int pinNum, boolean enable);

            public void handleSequencerEvent(SequencerEvent event, int arg);

            public void handleSync();

            public void handleAnalogOffline(int[] ADCResult, int numMsg);
      }

      class IncomingThread extends Thread {
            private List<Integer> analogPinValues_ = new ArrayList<Integer>();
            private List<Integer> analogFramePins_ = new ArrayList<Integer>();
            private List<Integer> newFramePins_ = new ArrayList<Integer>();
            private Set<Integer> removedPins_ = new HashSet<Integer>();
            private Set<Integer> addedPins_ = new HashSet<Integer>();

            private void calculateAnalogFrameDelta() {
                  removedPins_.clear();
                  removedPins_.addAll(analogFramePins_);
                  addedPins_.clear();
                  addedPins_.addAll(newFramePins_);
                  // Remove the intersection from both.
                  for (Iterator<Integer> it = removedPins_.iterator();
it.hasNext();) {
                        Integer current = it.next();
                        if (addedPins_.contains(current)) {
                              it.remove();
                              addedPins_.remove(current);
                        }
                  }
                  // swap
                  List<Integer> temp = analogFramePins_;
                  analogFramePins_ = newFramePins_;
                  newFramePins_ = temp;
            }

            private int readByte() throws IOException {
                  try {
                        int b = in_.read();
                        if (b < 0) {
                              throw new IOException("Unexpected stream
closure");
                        }

                        Log.v(TAG, "received: 0x" + Integer.toHexString(b));
                        return b;
```

```
                } catch (IOException e) {
                        Log.i(TAG, "IOIO disconnected");
                        throw e;
                }
        }

        private void readBytes(int size, byte[] buffer) throws IOException
{
                for (int i = 0; i < size; ++i) {
                        buffer[i] = (byte) readByte();
                }
        }

        @Override
        public void run() {
                super.run();
                setPriority(MAX_PRIORITY);
                int arg1;
                int arg2;
                int numPins;
                int size;
                byte[] data = new byte[256];
                try {
                        while (true) {
                                switch (arg1 = readByte()) {
                                case ESTABLISH_CONNECTION:
                                        if (readByte() != 'I' || readByte() != 'O'
|| readByte() != 'I'
                                                        || readByte() != 'O') {
                                                throw new IOException("Bad establish
connection magic");
                                        }
                                        byte[] hardwareId = new byte[8];
                                        byte[] bootloaderId = new byte[8];
                                        byte[] firmwareId = new byte[8];
                                        readBytes(8, hardwareId);
                                        readBytes(8, bootloaderId);
                                        readBytes(8, firmwareId);


      handler_.handleEstablishConnection(hardwareId, bootloaderId,
firmwareId);
                                        break;

                                case SOFT_RESET:
                                        analogFramePins_.clear();
                                        handler_.handleSoftReset();
                                        break;

                                case REPORT_DIGITAL_IN_STATUS:
                                        arg1 = readByte();
                                        handler_.handleReportDigitalInStatus(arg1
>> 2, (arg1 & 0x01) == 1);
                                        break;

                                case SET_CHANGE_NOTIFY:
                                        arg1 = readByte();
                                        handler_.handleSetChangeNotify(arg1 >> 2,
(arg1 & 0x01) == 1);
                                        break;

                                case REGISTER_PERIODIC_DIGITAL_SAMPLING:
```

```
                                    // TODO: implement
                                    break;

                              case REPORT_PERIODIC_DIGITAL_IN_STATUS:
                                    // TODO: implement
                                    break;

                              case REPORT_ANALOG_IN_FORMAT:
                                    numPins = readByte();
                                    newFramePins_.clear();
                                    for (int i = 0; i < numPins; ++i) {
                                          newFramePins_.add(readByte());
                                    }
                                    calculateAnalogFrameDelta();
                                    for (Integer i : removedPins_) {
                                          handler_.handleAnalogPinStatus(i,
false);
                                    }
                                    for (Integer i : addedPins_) {
                                          handler_.handleAnalogPinStatus(i,
true);
                                    }
                                    break;

                              case REPORT_ANALOG_IN_STATUS:
                                    numPins = analogFramePins_.size();
                                    int header = 0;
                                    analogPinValues_.clear();
                                    for (int i = 0; i < numPins; ++i) {
                                          if (i % 4 == 0) {
                                                header = readByte();
                                          }
                                          analogPinValues_.add((readByte() <<
2) | (header & 0x03));
                                          header >>= 2;
                                    }

      handler_.handleReportAnalogInStatus(analogFramePins_,
analogPinValues_);
                                    break;

                              case REPORT_ANALOG_OFFLINE_STATUS:
                                    // First, we read the number of messages
                                    int numMsg=readByte();
                                    Log.v(TAG, "Número de mensajes restantes:
" + numMsg);
                                    // Second, we read the number of samples
included in the message
                                    int numSamples=readByte();
                                    Log.v(TAG, "Número de muestras: " +
numSamples);
                                    // Array with the samples
                                    int[] ADCResult=new int[numSamples];
                                    // We have to read all the parameters of
the message
                                    // whose length is AnalogMsgLength
                                    for(int i=0; i<AnalogMsgLength;i++){
                                          if (i < numSamples) {
                                                arg1=readByte();  // LSB
                                                arg2=readByte();  // MSB
                                                ADCResult[i]=arg1 | (arg2<<8);
// Sample construction
```

```
                                                //Log.v(TAG, "Recibida
muestra: "+ ADCResult[i]);
                                          }
                                          else {
                                                // If the message is not full,
we read the rest of the message
                                                // in order to complete the
action.
                                                arg1=readByte();
                                                arg2=readByte();
                                          }
                                    }
                                    Log.v(TAG, "Muestras recibidas en mensaje
" + numMsg +  ": " + Arrays.toString(ADCResult));
                                    // We send the samples to the handler
                                    handler_.handleAnalogOffline(ADCResult,
numMsg);
                              break;

                              case UART_REPORT_TX_STATUS:
                                    arg1 = readByte();
                                    arg2 = readByte();
                                    handler_.handleUartReportTxStatus(arg1 &
0x03, (arg1 >> 2) | (arg2 << 6));
                                    break;

                              case UART_DATA:
                                    arg1 = readByte();
                                    size = (arg1 & 0x3F) + 1;
                                    readBytes(size, data);
                                    handler_.handleUartData(arg1 >> 6, size,
data);
                                    break;

                              case UART_STATUS:
                                    arg1 = readByte();
                                    if ((arg1 & 0x80) != 0) {
                                          handler_.handleUartOpen(arg1 &
0x03);
                                    } else {
                                          handler_.handleUartClose(arg1 &
0x03);
                                    }
                                    break;

                              case SPI_DATA:
                                    arg1 = readByte();
                                    arg2 = readByte();
                                    size = (arg1 & 0x3F) + 1;
                                    readBytes(size, data);
                                    handler_.handleSpiData(arg1 >> 6, arg2 &
0x3F, data, size);
                                    break;

                              case SPI_REPORT_TX_STATUS:
                                    arg1 = readByte();
                                    arg2 = readByte();
                                    handler_.handleSpiReportTxStatus(arg1 &
0x03, (arg1 >> 2) | (arg2 << 6));
                                    break;

                              case SPI_STATUS:
```

```
                                    arg1 = readByte();
                                    if ((arg1 & 0x80) != 0) {
                                          handler_.handleSpiOpen(arg1 & 0x03);
                                    } else {
                                          handler_.handleSpiClose(arg1 &
0x03);
                                    }
                                    break;

                              case I2C_STATUS:
                                    arg1 = readByte();
                                    if ((arg1 & 0x80) != 0) {
                                          handler_.handleI2cOpen(arg1 & 0x03);
                                    } else {
                                          handler_.handleI2cClose(arg1 &
0x03);
                                    }
                                    break;

                              case I2C_RESULT:
                                    arg1 = readByte();
                                    arg2 = readByte();
                                    if (arg2 != 0xFF) {
                                          readBytes(arg2, data);
                                    }
                                    handler_.handleI2cResult(arg1 & 0x03,
arg2, data);
                                    break;

                              case I2C_REPORT_TX_STATUS:
                                    arg1 = readByte();
                                    arg2 = readByte();
                                    handler_.handleI2cReportTxStatus(arg1 &
0x03, (arg1 >> 2) | (arg2 << 6));
                                    break;

                              case CHECK_INTERFACE_RESPONSE:
                                    arg1 = readByte();

      handler_.handleCheckInterfaceResponse((arg1 & 0x01) == 1);
                                    break;

                              case ICSP_REPORT_RX_STATUS:
                                    arg1 = readByte();
                                    arg2 = readByte();
                                    handler_.handleIcspReportRxStatus(arg1 |
(arg2 << 8));
                                    break;

                              case ICSP_RESULT:
                                    readBytes(2, data);
                                    handler_.handleIcspResult(2, data);
                                    break;

                              case ICSP_CONFIG:
                                    arg1 = readByte();
                                    if ((arg1 & 0x01) == 1) {
                                          handler_.handleIcspOpen();
                                    } else {
                                          handler_.handleIcspClose();
                                    }
                                    break;
```

```
                                case INCAP_STATUS:
                                      arg1 = readByte();
                                      if ((arg1 & 0x80) != 0) {
                                            handler_.handleIncapOpen(arg1 &
0x0F);
                                      } else {
                                            handler_.handleIncapClose(arg1 &
0x0F);
                                      }
                                      break;

                                case INCAP_REPORT:
                                      arg1 = readByte();
                                      size = arg1 >> 6;
                                      if (size == 0) {
                                            size = 4;
                                      }
                                      readBytes(size, data);
                                      handler_.handleIncapReport(arg1 & 0x0F,
size, data);
                                      break;

                                case SOFT_CLOSE:
                                      Log.d(TAG, "Received soft close.");
                                      throw new IOException("Soft close");

                                case CAPSENSE_REPORT:
                                      arg1 = readByte();
                                      arg2 = readByte();
                                      handler_.handleCapSenseReport(arg1 & 0x3F,
(arg1 >> 6) | (arg2 << 2));
                                      break;

                                case SET_CAPSENSE_SAMPLING:
                                      arg1 = readByte();
                                      handler_.handleSetCapSenseSampling(arg1 &
0x3F, (arg1 & 0x80) != 0);
                                      break;

                                case SEQUENCER_EVENT:
                                      arg1 = readByte();
                                      // OPEN and STOPPED events has an
additional argument.
                                      if (arg1 == 2 || arg1 == 4) {
                                            arg2 = readByte();
                                      } else {
                                            arg2 = 0;
                                      }
                                      try {

      handler_.handleSequencerEvent(SequencerEvent.values()[arg1], arg2);
                                      } catch (ArrayIndexOutOfBoundsException e)
{
                                            throw new IOException("Unexpected
eveent: " + arg1);
                                      }
                                      break;

                                case SYNC:
                                      handler_.handleSync();
                                      break;
```

```
                              default:
                                    throw new ProtocolError("Received
unexpected command: 0x"
                                                + Integer.toHexString(arg1));
                              }

                        }
                  } catch (IOException e) {
                        // This is the proper way to close -- nothing's wrong.
                  } catch (ProtocolError e) {
                        // This indicates invalid data coming in -- report the
error.
                        Log.e(TAG, "Protocol error: ", e);
                  } catch (Exception e) {
                        // This also probably indicates invalid data coming
in, which has been detected by
                        // the command handler -- report the error.
                        Log.e(TAG, "Protocol error: ", new ProtocolError(e));
                  } finally {
                        try {
                              in_.close();
                        } catch (IOException e) {
                        }
                        handler_.handleConnectionLost();
                  }
            }
      }

      private final InputStream in_;
      private final OutputStream out_;
      private final IncomingHandler handler_;
      private final IncomingThread thread_ = new IncomingThread();

      public IOIOProtocol(InputStream in, OutputStream out, IncomingHandler
handler) {
            in_ = in;
            out_ = out;
            handler_ = handler;
            thread_.start();
      }
}
```

IncomingState.java

```
package ioio.lib.impl;

import ioio.lib.api.exception.ConnectionLostException;
import ioio.lib.impl.Board.Hardware;
import ioio.lib.impl.IOIOProtocol.IncomingHandler;
import ioio.lib.impl.IOIOProtocol.SequencerEvent;
import ioio.lib.spi.Log;

import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Queue;
import java.util.Set;
import java.util.concurrent.ConcurrentLinkedQueue;

class IncomingState implements IncomingHandler {
	private static final String TAG = "IncomingState";

	enum ConnectionState {
		INIT, ESTABLISHED, CONNECTED, DISCONNECTED, UNSUPPORTED_IID
	}

	interface InputPinListener {
		void setValue(int value);
	}

	interface DisconnectListener {
		void disconnected();
	}

	interface DataModuleListener {
		void dataReceived(byte[] data, int size);

		void reportAdditionalBuffer(int bytesToAdd);
	}

	interface SyncListener {
		void sync();
	}

	interface SequencerEventListener {

		void opened(int arg);

		void nextCue();

		void paused();

		void stopped(int arg);

		void closed();

		void stalled();

	}

	class ListenerQueue<T> {
		private Queue<T> listeners_ = new ConcurrentLinkedQueue<T>();
		private boolean currentOpen_ = false;

		void pushListener(T listener) {
			listeners_.add(listener);
```

```
            }

            void closeCurrentListener() {
                  if (currentOpen_) {
                        currentOpen_ = false;
                        listeners_.remove();
                  }
            }

            void openNextListener() {
                  assert (!listeners_.isEmpty());
                  if (!currentOpen_) {
                        currentOpen_ = true;
                  }
            }

            T peek() {
                  assert (currentOpen_);
                  return listeners_.peek();
            }
      }

      class InputPinState extends ListenerQueue<InputPinListener> {
            void setValue(int v) {
                  peek().setValue(v);
            }
      }

      class DataModuleState extends ListenerQueue<DataModuleListener> {
            void dataReceived(byte[] data, int size) {
                  peek().dataReceived(data, size);
            }

            public void reportAdditionalBuffer(int bytesRemaining) {
                  peek().reportAdditionalBuffer(bytesRemaining);
            }
      }

      class SyncListeners extends ListenerQueue<SyncListener> {
            void syncRecieved() {
                  openNextListener();
                  peek().sync();
                  closeCurrentListener();
            }
      }

      private InputPinState[] intputPinStates_;
      private DataModuleState[] uartStates_;
      private DataModuleState[] twiStates_;
      private DataModuleState[] spiStates_;
      private DataModuleState[] incapStates_;
      private DataModuleState icspState_;
      private ListenerQueue<SequencerEventListener> sequencerListeners_;
      private SyncListeners syncListeners_;
      private final Set<DisconnectListener> disconnectListeners_ = new
HashSet<IncomingState.DisconnectListener>();
      private ConnectionState connection_ = ConnectionState.INIT;
      public String hardwareId_;
      public String bootloaderId_;
      public String firmwareId_;
      public Board board_;
      public int[] ADCResult_;
```

```
	public int numMsg_;

	synchronized public void waitConnectionEstablished()
			throws InterruptedException, ConnectionLostException {
		while (connection_ == ConnectionState.INIT) {
			wait();
		}
		if (connection_ == ConnectionState.DISCONNECTED) {
			throw new ConnectionLostException();
		}
	}

	synchronized public boolean waitForInterfaceSupport()
			throws InterruptedException, ConnectionLostException {
		if (connection_ == ConnectionState.INIT) {
			throw new IllegalStateException(
					"Have to connect before waiting for interface
support");
		}
		while (connection_ == ConnectionState.ESTABLISHED) {
			wait();
		}
		if (connection_ == ConnectionState.DISCONNECTED) {
			throw new ConnectionLostException();
		}
		return connection_ == ConnectionState.CONNECTED;
	}

	synchronized public void waitDisconnect() throws InterruptedException {
		while (connection_ != ConnectionState.DISCONNECTED) {
			wait();
		}
	}

	public void addInputPinListener(int pin, InputPinListener listener) {
		intputPinStates_[pin].pushListener(listener);
	}

	public void addUartListener(int uartNum, DataModuleListener listener) {
		uartStates_[uartNum].pushListener(listener);
	}

	public void addTwiListener(int twiNum, DataModuleListener listener) {
		twiStates_[twiNum].pushListener(listener);
	}

	public void addIncapListener(int incapNum, DataModuleListener listener)
{
		incapStates_[incapNum].pushListener(listener);
	}

	public void addIcspListener(DataModuleListener listener) {
		icspState_.pushListener(listener);
	}

	public void addSpiListener(int spiNum, DataModuleListener listener) {
		spiStates_[spiNum].pushListener(listener);
	}

	public void addSequencerEventListener(SequencerEventListener listener)
{
		sequencerListeners_.pushListener(listener);
```

```
	}

	public void addSyncListener(SyncListener listener) {
		syncListeners_.pushListener(listener);
	}

	synchronized public void addDisconnectListener(DisconnectListener
listener)
			throws ConnectionLostException {
		checkNotDisconnected();
		disconnectListeners_.add(listener);
	}

	synchronized public void removeDisconnectListener(
			DisconnectListener listener) {
		if (connection_ != ConnectionState.DISCONNECTED) {
			disconnectListeners_.remove(listener);
		}
	}

	@Override
	public synchronized void handleConnectionLost() {
		// logMethod("handleConnectionLost");
		connection_ = ConnectionState.DISCONNECTED;
		for (DisconnectListener listener : disconnectListeners_) {
			listener.disconnected();
		}
		disconnectListeners_.clear();
		notifyAll();
	}

	@Override
	public void handleSoftReset() {
		// logMethod("handleSoftReset");
		for (InputPinState pinState : intputPinStates_) {
			pinState.closeCurrentListener();
		}
		for (DataModuleState uartState : uartStates_) {
			uartState.closeCurrentListener();
		}
		for (DataModuleState twiState : twiStates_) {
			twiState.closeCurrentListener();
		}
		for (DataModuleState spiState : spiStates_) {
			spiState.closeCurrentListener();
		}
		for (DataModuleState incapState : incapStates_) {
			incapState.closeCurrentListener();
		}
		icspState_.closeCurrentListener();
		sequencerListeners_.closeCurrentListener();
	}

	@Override
	synchronized public void handleCheckInterfaceResponse(boolean
supported) {
		// logMethod("handleCheckInterfaceResponse", supported);
		connection_ = supported ? ConnectionState.CONNECTED
				: ConnectionState.UNSUPPORTED_IID;
		notifyAll();
	}
```

```
	@Override
	public void handleSetChangeNotify(int pin, boolean changeNotify) {
		// logMethod("handleSetChangeNotify", pin, changeNotify);
		if (changeNotify) {
			intputPinStates_[pin].openNextListener();
		} else {
			intputPinStates_[pin].closeCurrentListener();
		}
	}

	@Override
	public void handleRegisterPeriodicDigitalSampling(int pin, int
freqScale) {
		// logMethod("handleRegisterPeriodicDigitalSampling", pin,
freqScale);
		assert (false);
	}

	@Override
	public void handleAnalogPinStatus(int pin, boolean open) {
		// logMethod("handleAnalogPinStatus", pin, open);
		if (open) {
			intputPinStates_[pin].openNextListener();
		} else {
			intputPinStates_[pin].closeCurrentListener();
		}
	}

	@Override
	public void handleUartData(int uartNum, int numBytes, byte[] data) {
		// logMethod("handleUartData", uartNum, numBytes, data);
		uartStates_[uartNum].dataReceived(data, numBytes);
	}

	@Override
	public void handleUartOpen(int uartNum) {
		// logMethod("handleUartOpen", uartNum);
		uartStates_[uartNum].openNextListener();
	}

	@Override
	public void handleUartClose(int uartNum) {
		// logMethod("handleUartClose", uartNum);
		uartStates_[uartNum].closeCurrentListener();
	}

	@Override
	public void handleSpiOpen(int spiNum) {
		// logMethod("handleSpiOpen", spiNum);
		spiStates_[spiNum].openNextListener();
	}

	@Override
	public void handleSpiClose(int spiNum) {
		// logMethod("handleSpiClose", spiNum);
		spiStates_[spiNum].closeCurrentListener();
	}

	@Override
	public void handleI2cOpen(int i2cNum) {
		// logMethod("handleI2cOpen", i2cNum);
		twiStates_[i2cNum].openNextListener();
```

```
	}

	@Override
	public void handleI2cClose(int i2cNum) {
		// logMethod("handleI2cClose", i2cNum);
		twiStates_[i2cNum].closeCurrentListener();
	}

	@Override
	public void handleIcspOpen() {
		// logMethod("handleIcspOpen");
		icspState_.openNextListener();
	}

	@Override
	public void handleIcspClose() {
		// logMethod("handleIcspClose");
		icspState_.closeCurrentListener();
	}

	@Override
	public void handleEstablishConnection(byte[] hardwareId,
			byte[] bootloaderId, byte[] firmwareId) {
		hardwareId_ = new String(hardwareId);
		bootloaderId_ = new String(bootloaderId);
		firmwareId_ = new String(firmwareId);

		Log.i(TAG, "IOIO Connection established. Hardware ID: " +
hardwareId_
				+ " Bootloader ID: " + bootloaderId_ + " Firmware ID:
"
				+ firmwareId_);
		try {
			board_ = Board.valueOf(hardwareId_);
		} catch (IllegalArgumentException e) {
			Log.e(TAG, "Unknown board: " + hardwareId_);
		}
		if (board_ != null) {
			final Hardware hw = board_.hardware;
			intputPinStates_ = new InputPinState[hw.numPins()];
			for (int i = 0; i < intputPinStates_.length; ++i) {
				intputPinStates_[i] = new InputPinState();
			}
			uartStates_ = new DataModuleState[hw.numUartModules()];
			for (int i = 0; i < uartStates_.length; ++i) {
				uartStates_[i] = new DataModuleState();
			}
			twiStates_ = new DataModuleState[hw.numTwiModules()];
			for (int i = 0; i < twiStates_.length; ++i) {
				twiStates_[i] = new DataModuleState();
			}
			spiStates_ = new DataModuleState[hw.numSpiModules()];
			for (int i = 0; i < spiStates_.length; ++i) {
				spiStates_[i] = new DataModuleState();
			}
			incapStates_ = new DataModuleState[2
					* hw.incapDoubleModules().length
					+ hw.incapSingleModules().length];
			for (int i = 0; i < incapStates_.length; ++i) {
				incapStates_[i] = new DataModuleState();
			}
			icspState_ = new DataModuleState();
```

```
                  sequencerListeners_ = new
ListenerQueue<SequencerEventListener>();
                  syncListeners_ = new SyncListeners();
            }
            synchronized (this) {
                  connection_ = ConnectionState.ESTABLISHED;
                  notifyAll();
            }
      }

      @Override
      public void handleUartReportTxStatus(int uartNum, int bytesRemaining) {
            // logMethod("handleUartReportTxStatus", uartNum, bytesRemaining);
            uartStates_[uartNum].reportAdditionalBuffer(bytesRemaining);
      }

      @Override
      public void handleI2cReportTxStatus(int i2cNum, int bytesRemaining) {
            // logMethod("handleI2cReportTxStatus", i2cNum, bytesRemaining);
            twiStates_[i2cNum].reportAdditionalBuffer(bytesRemaining);
      }

      @Override
      public void handleSpiData(int spiNum, int ssPin, byte[] data, int
dataBytes) {
            // logMethod("handleSpiData", spiNum, ssPin, data, dataBytes);
            spiStates_[spiNum].dataReceived(data, dataBytes);
      }

      @Override
      public void handleIcspReportRxStatus(int bytesRemaining) {
            // logMethod("handleIcspReportRxStatus", bytesRemaining);
            icspState_.reportAdditionalBuffer(bytesRemaining);
      }

      @Override
      public void handleReportDigitalInStatus(int pin, boolean level) {
            // logMethod("handleReportDigitalInStatus", pin, level);
            intputPinStates_[pin].setValue(level ? 1 : 0);
      }

      @Override
      public void handleReportPeriodicDigitalInStatus(int frameNum,
                  boolean[] values) {
            // logMethod("handleReportPeriodicDigitalInStatus", frameNum,
values);
      }

      @Override
      public void handleReportAnalogInStatus(List<Integer> pins,
                  List<Integer> values) {
            // logMethod("handleReportAnalogInStatus", pins, values);
            for (int i = 0; i < pins.size(); ++i) {
                  intputPinStates_[pins.get(i)].setValue(values.get(i));
            }
      }

      @Override
      public void handleSpiReportTxStatus(int spiNum, int bytesRemaining) {
            // logMethod("handleSpiReportTxStatus", spiNum, bytesRemaining);
            spiStates_[spiNum].reportAdditionalBuffer(bytesRemaining);
      }
```

```
@Override
public void handleI2cResult(int i2cNum, int size, byte[] data) {
	// logMethod("handleI2cResult", i2cNum, size, data);
	twiStates_[i2cNum].dataReceived(data, size);
}

@Override
public void handleIncapReport(int incapNum, int size, byte[] data) {
	// logMethod("handleIncapReport", incapNum, size, data);
	incapStates_[incapNum].dataReceived(data, size);
}

@Override
public void handleIncapClose(int incapNum) {
	// logMethod("handleIncapClose", incapNum);
	incapStates_[incapNum].closeCurrentListener();
}

@Override
public void handleIncapOpen(int incapNum) {
	// logMethod("handleIncapOpen", incapNum);
	incapStates_[incapNum].openNextListener();
}

@Override
public void handleIcspResult(int size, byte[] data) {
	// logMethod("handleIcspResult", size, data);
	icspState_.dataReceived(data, size);
}

@Override
public void handleCapSenseReport(int pinNum, int value) {
	// logMethod("handleCapSenseReport", pinNum, value);
	intputPinStates_[pinNum].setValue(value);
}

@Override
public void handleSetCapSenseSampling(int pinNum, boolean enable) {
	// logMethod("handleSetCapSenseSampling", pinNum, enable);
	if (enable) {
		intputPinStates_[pinNum].openNextListener();
	} else {
		intputPinStates_[pinNum].closeCurrentListener();
	}
}

@Override
public void handleSequencerEvent(SequencerEvent event, int arg) {
	// logMethod("handleSequencerEvent", event, arg);
	switch (event) {
	case OPENED:
		sequencerListeners_.openNextListener();
		sequencerListeners_.peek().opened(arg);
		break;

	case NEXT_CUE:
		sequencerListeners_.peek().nextCue();
		break;

	case PAUSED:
		sequencerListeners_.peek().paused();
```

```
                  break;

            case STOPPED:
                  sequencerListeners_.peek().stopped(arg);
                  break;

            case CLOSED:
                  sequencerListeners_.peek().closed();
                  sequencerListeners_.closeCurrentListener();
                  break;

            case STALLED:
                  sequencerListeners_.peek().stalled();
            }
      }


      @Override
      public void handleSync() {
            syncListeners_.syncRecieved();
      }

      @Override
      public void handleAnalogOffline(int[] ADCResult, int numMsg) {
            ADCResult_=ADCResult;
            numMsg_=numMsg;
            Log.v(TAG, "Mensajes restantes (handler): "+ numMsg);
            Log.v(TAG, "Muestras recibidas (handler): "+
Arrays.toString(ADCResult_));
      }

      private void checkNotDisconnected() throws ConnectionLostException {
            if (connection_ == ConnectionState.DISCONNECTED) {
                  throw new ConnectionLostException();
            }
      }
}
```

IOIO.java

```
package ioio.lib.api;

import ioio.lib.api.PulseInput.PulseMode;
import ioio.lib.api.TwiMaster.Rate;
import ioio.lib.api.Uart.Parity;
import ioio.lib.api.Uart.StopBits;
import ioio.lib.api.exception.ConnectionLostException;
import ioio.lib.api.exception.IncompatibilityException;
import ioio.lib.api.exception.OutOfResourceException;

import java.io.Closeable;

/**
 * This interface provides control over all the IOIO board functions.
 * <p>
 * An instance of this interface is typically obtained by using the {@link
IOIOFactory} class.
 * Initially, a connection should be established, by calling {@link
#waitForConnect()}. This method
```

```
 * will block until the board is connected an a connection has been
established.
 * <p>
 * During the connection process, this library verifies that the IOIO
firmware is compatible with
 * the required version. If not, {@link #waitForConnect()} will throw a
 * {@link IncompatibilityException}, putting the {@link IOIO} instance in a
"zombie" state: nothing
 * could be done with it except calling {@link #disconnect()}, or waiting for
the physical
 * connection to drop via {@link #waitForDisconnect()}.
 * <p>
 * As soon as a connection is established, the IOIO can be used, typically,
by calling the openXXX()
 * functions to obtain additional interfaces for controlling specific
function of the board.
 * <p>
 * Whenever a connection is lost as a result of physically disconnecting the
board or as a result of
 * calling {@link #disconnect()}, this instance and all the interfaces
obtained from it become
 * invalid, and will throw a {@link ConnectionLostException} on every
operation. Once the connection
 * is lost, those instances cannot be recycled, but rather it is required to
create new ones and
 * wait for a connection again.
 * <p>
 * Initially all pins are tri-stated (floating), and all functions are
disabled. Whenever a
 * connection is lost or dropped, the board will immediately return to the
this initial state.
 * <p>
 * Typical usage:
 *
 * <pre>
 * IOIO ioio = IOIOFactory.create();
 * try {
 *   ioio.waitForConnect();
 *   DigitalOutput out = ioio.openDigitalOutput(10);
 *   out.write(true);
 *   ...
 * } catch (ConnectionLostException e) {
 * } catch (Exception e) {
 *   ioio.disconnect();
 * } finally {
 *   ioio.waitForDisconnect();
 * }
 * </pre>
 *
 * @see IOIOFactory#create()
 */
public interface IOIO {
      /** An invalid pin number. */
      public static final int INVALID_PIN = -1;
      /** The pin number used to designate the on-board 'stat' LED. */
      public static final int LED_PIN = 0;

      /**
       * A versioned component in the system.
       *
       * @see IOIO#getImplVersion(VersionType)
       */
```

```
	public enum VersionType {
		/** Hardware version. */
		HARDWARE_VER,
		/** Bootloader version. */
		BOOTLOADER_VER,
		/** Application layer firmware version. */
		APP_FIRMWARE_VER,
		/** IOIOLib version. */
		IOIOLIB_VER
	}

	/**
	 * A state of a IOIO instance.
	 */
	public enum State {
		/** Connection not yet established. */
		INIT,
		/** Connected. */
		CONNECTED,
		/** Connection established, incompatible firmware detected. */
		INCOMPATIBLE,
		/** Disconnected. Instance is useless. */
		DEAD
	}

	/**
	 * Establishes connection with the IOIO board.
	 * <p>
	 * This method is blocking until connection is established. This method
can be aborted by
	 * calling {@link #disconnect()}. In this case, it will throw a {@link
ConnectionLostException}.
	 *
	 * @throws ConnectionLostException
	 *             An error occurred during connection or disconnect() has
been called during
	 *             connection. The instance state is disconnected.
	 * @throws IncompatibilityException
	 *             An incompatible board firmware of hardware has been
detected. The instance state
	 *             is disconnected.
	 * @see #disconnect()
	 * @see #waitForDisconnect()
	 */
	public void waitForConnect() throws ConnectionLostException,
IncompatibilityException;

	/**
	 * Closes the connection to the board, or aborts a connection process
started with
	 * waitForConnect().
	 * <p>
	 * Once this method is called, this IOIO instance and all the instances
obtain from it become
	 * invalid and will throw an exception on every operation.
	 * <p>
	 * This method is asynchronous, i.e. it returns immediately, but it is
not guaranteed that all
	 * connection-related resources has already been freed and can be
reused upon return. In cases
	 * when this is important, client can call {@link
#waitForDisconnect()}, which will block until
```

```
         * all resources have been freed.
         */
        public void disconnect();

        /**
         * Blocks until IOIO has been disconnected and all connection-related
resources have been freed,
         * so that a new connection can be attempted.
         *
         * @throws InterruptedException
         *             When interrupt() has been called on this thread. This
might mean that an
         *             immediate attempt to create and connect a new IOIO
object might fail for resource
         *             contention.
         * @see #disconnect()
         * @see #waitForConnect()
         */
        public void waitForDisconnect() throws InterruptedException;

        /**
         * Gets the connections state.
         *
         * @return The connection state.
         */
        public State getState();

        /**
         * Resets the entire state (returning to initial state), without
dropping the connection.
         * <p>
         * It is equivalent to calling {@link Closeable#close()} on every
interface obtained from this
         * instance. A connection must have been established prior to calling
this method, by invoking
         * {@link #waitForConnect()}.
         *
         * @throws ConnectionLostException
         *             Connection was lost before or during the execution of
this method.
         * @see #hardReset()
         */
        public void softReset() throws ConnectionLostException;

        /**
         * Equivalent to disconnecting and reconnecting the board power supply.
         * <p>
         * The connection will be dropped and not reestablished. Full boot
sequence will take place, so
         * firmware upgrades can be performed. A connection must have been
established prior to calling
         * this method, by invoking {@link #waitForConnect()}.
         *
         * @throws ConnectionLostException
         *             Connection was lost before or during the execution of
this method.
         * @see #softReset()
         */
        public void hardReset() throws ConnectionLostException;

        /**
         * Query the implementation version of the system's components. The
```

```
implementation version
       * uniquely identifies a hardware revision or a software build.
Returned version IDs are always
       * 8-character long, according to the IOIO versioning system: first 4
characters are the version
       * authority and last 4 characters are the revision.
       *
       * @param v
       *            The component whose version we query.
       * @return An 8-character implementation version ID.
       */
      public String getImplVersion(VersionType v);

      /**
       * Open a pin for digital input.
       * <p>
       * A digital input pin can be used to read logic-level signals. The pin
will operate in this
       * mode until close() is invoked on the returned interface. It is
illegal to open a pin that has
       * already been opened and has not been closed. A connection must have
been established prior to
       * calling this method, by invoking {@link #waitForConnect()}.
       *
       * @param spec
       *            Pin specification, consisting of the pin number, as
labeled on the board, and the
       *            mode, which determines whether the pin will be floating,
pull-up or pull-down. See
       *            {@link DigitalInput.Spec.Mode} for more information.
       * @return Interface of the assigned pin.
       * @throws ConnectionLostException
       *             Connection was lost before or during the execution of
this method.
       * @see DigitalInput
       */
      public DigitalInput openDigitalInput(DigitalInput.Spec spec) throws
ConnectionLostException;

      /**
       * Shorthand for openDigitalInput(new DigitalInput.Spec(pin)).
       *
       * @see #openDigitalInput(ioio.lib.api.DigitalInput.Spec)
       */
      public DigitalInput openDigitalInput(int pin) throws
ConnectionLostException;

      /**
       * Shorthand for openDigitalInput(new DigitalInput.Spec(pin, mode)).
       *
       * @see #openDigitalInput(ioio.lib.api.DigitalInput.Spec)
       */
      public DigitalInput openDigitalInput(int pin, DigitalInput.Spec.Mode
mode)
                  throws ConnectionLostException;

      /**
       * Open a pin for digital output.
       * <p>
       * A digital output pin can be used to generate logic-level signals.
The pin will operate in
       * this mode until close() is invoked on the returned interface. It is
```

```
illegal to open a pin
       * that has already been opened and has not been closed. A connection
must have been established
       * prior to calling this method, by invoking {@link #waitForConnect()}.
       *
       * @param spec
       *            Pin specification, consisting of the pin number, as
labeled on the board, and the
       *            mode, which determines whether the pin will be normal or
open-drain. See
       *            {@link DigitalOutput.Spec.Mode} for more information.
       * @param startValue
       *            The initial logic level this pin will generate as soon at
it is open.
       * @return Interface of the assigned pin.
       * @throws ConnectionLostException
       *             Connection was lost before or during the execution of
this method.
       * @see DigitalOutput
       */
      public DigitalOutput openDigitalOutput(DigitalOutput.Spec spec, boolean
startValue)
                  throws ConnectionLostException;

      /**
       * Shorthand for openDigitalOutput(new DigitalOutput.Spec(pin, mode),
startValue).
       *
       * @see #openDigitalOutput(ioio.lib.api.DigitalOutput.Spec, boolean)
       */
      public DigitalOutput openDigitalOutput(int pin, DigitalOutput.Spec.Mode
mode, boolean startValue)
                  throws ConnectionLostException;

      /**
       * Shorthand for openDigitalOutput(new DigitalOutput.Spec(pin),
startValue). Pin mode will be
       * "normal" (as opposed to "open-drain".
       *
       * @see #openDigitalOutput(ioio.lib.api.DigitalOutput.Spec, boolean)
       */
      public DigitalOutput openDigitalOutput(int pin, boolean startValue)
                  throws ConnectionLostException;

      /**
       * Shorthand for openDigitalOutput(new DigitalOutput.Spec(pin), false).
Pin mode will be
       * "normal" (as opposed to "open-drain".
       *
       * @see #openDigitalOutput(ioio.lib.api.DigitalOutput.Spec, boolean)
       */
      public DigitalOutput openDigitalOutput(int pin) throws
ConnectionLostException;


      /**
       * Open a pin for analog input.
       * <p>
       * An analog input pin can be used to measure voltage. Note that not
every pin can be used as an
       * analog input. See board documentation for the legal pins and
permitted voltage range.
```

```
       * <p>
       * The pin will operate in this mode until close() is invoked on the
returned interface. It is
       * illegal to open a pin that has already been opened and has not been
closed. A connection must
       * have been established prior to calling this method, by invoking
{@link #waitForConnect()}.
       *
       * @param pin
       *            Pin number, as labeled on the board.
       * @return Interface of the assigned pin.
       * @throws ConnectionLostException
       *             Connection was lost before or during the execution of
this method.
       * @see AnalogInput
       */
      public AnalogInput openAnalogInput(int pin) throws
ConnectionLostException;

      /**
       * Open a pin for PWM (Pulse-Width Modulation) output.
       * <p>
       * A PWM pin produces a logic-level PWM signal. These signals are
typically used for simulating
       * analog outputs for controlling the intensity of LEDs, the rotation
speed of motors, etc. They
       * are also frequently used for controlling hobby servo motors.
       * <p>
       * Note that not every pin can be used as PWM output. In addition, the
total number of
       * concurrent PWM modules in use is limited. See board documentation
for the legal pins and
       * limit on concurrent usage.
       * <p>
       * The pin will operate in this mode until close() is invoked on the
returned interface. It is
       * illegal to open a pin that has already been opened and has not been
closed. A connection must
       * have been established prior to calling this method, by invoking
{@link #waitForConnect()}.
       *
       * @param spec
       *            Pin specification, consisting of the pin number, as
labeled on the board, and the
       *            mode, which determines whether the pin will be normal or
open-drain. See
       *            {@link DigitalOutput.Spec.Mode} for more information.
       * @param freqHz
       *            PWM frequency, in Hertz.
       * @return Interface of the assigned pin.
       * @throws ConnectionLostException
       *             Connection was lost before or during the execution of
this method.
       * @throws OutOfResourceException
       *             This is a runtime exception, so it is not necessary to
catch it if the client
       *             guarantees that the total number of concurrent PWM
resources is not exceeded.
       * @see PwmOutput
       */
      public PwmOutput openPwmOutput(DigitalOutput.Spec spec, int freqHz)
                  throws ConnectionLostException;
```

```
      /**
       * Shorthand for openPwmOutput(new DigitalOutput.Spec(pin), freqHz).
       *
       * @see #openPwmOutput(ioio.lib.api.DigitalOutput.Spec, int)
       */
      public PwmOutput openPwmOutput(int pin, int freqHz) throws
ConnectionLostException;

      /**
       * Open a pin for pulse input.
       * <p>
       * The pulse input module is quite flexible. It enables several kinds
of timing measurements on
       * a digital signal: pulse width measurement (positive or negative
pulse), and frequency of a
       * periodic signal.
       * <p>
       * Note that not every pin can be used as pulse input. In addition, the
total number of
       * concurrent pulse input modules in use is limited. See board
documentation for the legal pins
       * and limit on concurrent usage.
       * <p>
       * The pin will operate in this mode until close() is invoked on the
returned interface. It is
       * illegal to open a pin that has already been opened and has not been
closed. A connection must
       * have been established prior to calling this method, by invoking
{@link #waitForConnect()}.
       *
       * @param spec
       *            Pin specification, consisting of the pin number, as
labeled on the board, and the
       *            mode, which determines whether the pin will be floating,
pull-up or pull-down. See
       *            {@link DigitalInput.Spec.Mode} for more information.
       * @param rate
       *            The clock rate to use for timing the signal. A faster
clock rate will result in
       *            better precision but will only be able to measure narrow
pulses / high
       *            frequencies.
       * @param mode
       *            The mode in which to operate. Determines whether the
module will measure pulse
       *            durations or frequency.
       * @param doublePrecision
       *            Whether to open a double-precision pulse input module.
Double- precision modules
       *            enable reading of much longer pulses and lower
frequencies with high accuracy than
       *            single precision modules. However, their number is
limited, so when possible, and
       *            if the resources are all needed, use single-precision.
For more details on the
       *            exact spec of single- vs. double- precision, see {@link
PulseInput}.
       * @return An instance of the {@link PulseInput}, which can be used to
obtain the data.
       * @throws ConnectionLostException
       *             Connection was lost before or during the execution of
```

```
this method.
       * @throws OutOfResourceException
       *             This is a runtime exception, so it is not necessary to
catch it if the client
       *             guarantees that the total number of concurrent PWM
resources is not exceeded.
       * @see PulseInput
       */
      public PulseInput openPulseInput(DigitalInput.Spec spec,
PulseInput.ClockRate rate,
                  PulseInput.PulseMode mode, boolean doublePrecision) throws
ConnectionLostException;

      /**
       * Shorthand for openPulseInput(new DigitalInput.Spec(pin), rate, mode,
true), i.e. opens a
       * double-precision, 16MHz pulse input on the given pin with the given
mode.
       *
       * @see #openPulseInput(ioio.lib.api.DigitalInput.Spec,
ioio.lib.api.PulseInput.ClockRate,
       *      ioio.lib.api.PulseInput.PulseMode, boolean)
       */
      public PulseInput openPulseInput(int pin, PulseMode mode) throws
ConnectionLostException;

      /**
       * Open a UART module, enabling a bulk transfer of byte buffers.
       * <p>
       * UART is a very common hardware communication protocol, enabling
full- duplex, asynchronous
       * point-to-point data transfer. It typically serves for opening
consoles or as a basis for
       * higher-level protocols, such as MIDI RS-232, and RS-485.
       * <p>
       * Note that not every pin can be used for UART RX or TX. In addition,
the total number of
       * concurrent UART modules in use is limited. See board documentation
for the legal pins and
       * limit on concurrent usage.
       * <p>
       * The UART module will operate, and the pins will work in their
respective modes until close()
       * is invoked on the returned interface. It is illegal to use pins that
have already been opened
       * and has not been closed. A connection must have been established
prior to calling this
       * method, by invoking {@link #waitForConnect()}.
       *
       * @param rx
       *            Pin specification for the RX pin, consisting of the pin
number, as labeled on the
       *            board, and the mode, which determines whether the pin
will be floating, pull-up or
       *            pull-down. See {@link DigitalInput.Spec.Mode} for more
information. null can be
       *            passed to designate that we do not want RX input to this
module.
       * @param tx
       *            Pin specification for the TX pin, consisting of the pin
number, as labeled on the
       *            board, and the mode, which determines whether the pin
```

```
will be normal or
       *            open-drain. See {@link DigitalOutput.Spec.Mode} for more
information. null can be
       *            passed to designate that we do not want TX output to this
module.
       * @param baud
       *            The clock frequency of the UART module in Hz.
       * @param parity
       *            The parity mode, as in {@link Parity}.
       * @param stopbits
       *            Number of stop bits, as in {@link StopBits}.
       * @return Interface of the assigned module.
       * @throws ConnectionLostException
       *             Connection was lost before or during the execution of
this method.
       * @throws OutOfResourceException
       *             This is a runtime exception, so it is not necessary to
catch it if the client
       *             guarantees that the total number of concurrent UART
resources is not exceeded.
       * @see Uart
       */
      public Uart openUart(DigitalInput.Spec rx, DigitalOutput.Spec tx, int
baud, Parity parity,
                  StopBits stopbits) throws ConnectionLostException;

      /**
       * Shorthand for
       * {@link #openUart(DigitalInput.Spec, DigitalOutput.Spec, int,
Uart.Parity, Uart.StopBits)} ,
       * where the input pins use their default specs. {@link #INVALID_PIN}
can be used on either pin
       * if a TX- or RX-only UART is needed.
       *
       * @see #openUart(DigitalInput.Spec, DigitalOutput.Spec, int,
Uart.Parity, Uart.StopBits)
       */
      public Uart openUart(int rx, int tx, int baud, Parity parity, StopBits
stopbits)
                  throws ConnectionLostException;

      /**
       * Open a SPI master module, enabling communication with multiple SPI-
enabled slave modules.
       * <p>
       * SPI is a common hardware communication protocol, enabling full-
duplex, synchronous
       * point-to-multi-point data transfer. It requires MOSI, MISO and CLK
lines shared by all nodes,
       * as well as a SS line per slave, connected between this slave and a
respective pin on the
       * master. The MISO line should operate in pull-up mode, using either
the internal pull-up or an
       * external resistor.
       * <p>
       * Note that not every pin can be used for SPI MISO, MOSI or CLK. In
addition, the total number
       * of concurrent SPI modules in use is limited. See board documentation
for the legal pins and
       * limit on concurrent usage.
       * <p>
       * The SPI module will operate, and the pins will work in their
```

```
respective modes until close()
       * is invoked on the returned interface. It is illegal to use pins that
have already been opened
       * and has not been closed. A connection must have been established
prior to calling this
       * method, by invoking {@link #waitForConnect()}.
       *
       * @param miso
       *            Pin specification for the MISO (Master In Slave Out) pin,
consisting of the pin
       *            number, as labeled on the board, and the mode, which
determines whether the pin
       *            will be floating, pull-up or pull-down. See {@link
DigitalInput.Spec.Mode} for
       *            more information.
       * @param mosi
       *            Pin specification for the MOSI (Master Out Slave In) pin,
consisting of the pin
       *            number, as labeled on the board, and the mode, which
determines whether the pin
       *            will be normal or open-drain. See {@link
DigitalOutput.Spec.Mode} for more
       *            information.
       * @param clk
       *            Pin specification for the CLK pin, consisting of the pin
number, as labeled on the
       *            board, and the mode, which determines whether the pin
will be normal or
       *            open-drain. See {@link DigitalOutput.Spec.Mode} for more
information.
       * @param slaveSelect
       *            An array of pin specifications for each of the slaves' SS
(Slave Select) pin. The
       *            index of this array designates the slave index, used
later to refer to this slave.
       *            The spec is consisting of the pin number, as labeled on
the board, and the mode,
       *            which determines whether the pin will be normal or open-
drain. See
       *            {@link DigitalOutput.Spec.Mode} for more information.
       * @param config
       *            The configuration of the SPI module. See {@link
SpiMaster.Config} for details.
       * @return Interface of the assigned module.
       * @throws ConnectionLostException
       *             Connection was lost before or during the execution of
this method.
       * @throws OutOfResourceException
       *             This is a runtime exception, so it is not necessary to
catch it if the client
       *             guarantees that the total number of concurrent SPI
resources is not exceeded.
       * @see SpiMaster
       */
      public SpiMaster openSpiMaster(DigitalInput.Spec miso,
DigitalOutput.Spec mosi,
                  DigitalOutput.Spec clk, DigitalOutput.Spec[] slaveSelect,
SpiMaster.Config config)
                  throws ConnectionLostException;

      /**
       * Shorthand for
```

```
       * {@link #openSpiMaster(ioio.lib.api.DigitalInput.Spec,
ioio.lib.api.DigitalOutput.Spec, ioio.lib.api.DigitalOutput.Spec,
ioio.lib.api.DigitalOutput.Spec[], ioio.lib.api.SpiMaster.Config)}
       * , where the pins are all open with the default modes and default
configuration values are
       * used.
       *
       * @see #openSpiMaster(ioio.lib.api.DigitalInput.Spec,
ioio.lib.api.DigitalOutput.Spec,
       *      ioio.lib.api.DigitalOutput.Spec,
ioio.lib.api.DigitalOutput.Spec[],
       *      ioio.lib.api.SpiMaster.Config)
       */
      public SpiMaster openSpiMaster(int miso, int mosi, int clk, int[]
slaveSelect,
                  SpiMaster.Rate rate) throws ConnectionLostException;

      /**
       * Shorthand for
       * {@link #openSpiMaster(ioio.lib.api.DigitalInput.Spec,
ioio.lib.api.DigitalOutput.Spec, ioio.lib.api.DigitalOutput.Spec,
ioio.lib.api.DigitalOutput.Spec[], ioio.lib.api.SpiMaster.Config)}
       * , where the MISO pins is opened with pull up, and the other pins are
open with the default
       * modes and default configuration values are used. In this version, a
single slave is used.
       *
       * @see #openSpiMaster(ioio.lib.api.DigitalInput.Spec,
ioio.lib.api.DigitalOutput.Spec,
       *      ioio.lib.api.DigitalOutput.Spec,
ioio.lib.api.DigitalOutput.Spec[],
       *      ioio.lib.api.SpiMaster.Config)
       */
      public SpiMaster openSpiMaster(int miso, int mosi, int clk, int
slaveSelect, SpiMaster.Rate rate)
                  throws ConnectionLostException;

      /**
       * Open a TWI (Two-Wire Interface, such as I2C/SMBus) master module,
enabling communication with
       * multiple TWI-enabled slave modules.
       * <p>
       * TWI is a common hardware communication protocol, enabling half-
duplex, synchronous
       * point-to-multi-point data transfer. It requires a physical
connection of two lines (SDA, SCL)
       * shared by all the bus nodes, where the SDA is open-drain and
externally pulled-up.
       * <p>
       * Note that there is a fixed number of TWI modules, and the pins they
use are static. Client
       * has to make sure these pins are not already opened before calling
this method. See board
       * documentation for the number of modules and the respective pins they
use.
       * <p>
       * The TWI module will operate, and the pins will work in their
respective modes until close()
       * is invoked on the returned interface. It is illegal to use pins that
have already been opened
       * and has not been closed. A connection must have been established
prior to calling this
```

```
         * method, by invoking {@link #waitForConnect()}.
         *
         * @param twiNum
         *            The TWI module index to use. Will also determine the pins
used.
         * @param rate
         *            The clock rate. Can be 100KHz / 400KHz / 1MHz.
         * @param smbus
         *            When true, will use SMBus voltage levels. When false, I2C
voltage levels.
         * @return Interface of the assigned module.
         * @throws ConnectionLostException
         *             Connection was lost before or during the execution of
this method.
         * @see TwiMaster
         */
        public TwiMaster openTwiMaster(int twiNum, Rate rate, boolean smbus)
                throws ConnectionLostException;

        /**
         * Open an ICSP channel, enabling Flash programming of an external PIC
MCU, and in particular,
         * another IOIO board.
         * <p>
         * ICSP (In-Circuit Serial Programming) is a protocol intended for
programming of PIC MCUs. It
         * is a serial protocol over three wires: PGC (clock), PGD (data) and
MCLR (reset), where PGC
         * and MCLR are controlled by the master and PGD is shared by the
master and slave, depending on
         * the transaction state.
         * <p>
         * Note that there is only one ICSP modules, and the pins it uses are
static. Client has to make
         * sure that the ICSP module is not already in use, as well as those
dedicated pins. See board
         * documentation for the actual pins used for ICSP.
         *
         * @return Interface of the ICSP module.
         * @see IcspMaster
         * @throws ConnectionLostException
         *             Connection was lost before or during the execution of
this method.
         */
        public IcspMaster openIcspMaster() throws ConnectionLostException;

        /**
         * Shorthand for openCapSense(pin, CapSense.DEFAULT_COEF).
         *
         * @see #openCapSense(int, float)
         */
        public CapSense openCapSense(int pin) throws ConnectionLostException;

        /**
         * Open a pin for cap-sense.
         * <p>
         * A cap-sense input pin can be used to measure capacitance, typically
in touch sensing
         * applications. Note that not every pin can be used as cap- sense. See
board documentation for
         * the legal pins.
         * <p>
```

```
       * The pin will operate in this mode until close() is invoked on the
returned interface. It is
       * illegal to open a pin that has already been opened and has not been
closed. A connection must
       * have been established prior to calling this method, by invoking
{@link #waitForConnect()}.
       *
       * @param pin
       *            Pin number, as labeled on the board.
       * @return Interface of the assigned pin.
       * @throws ConnectionLostException
       *             Connection was lost before or during the execution of
this method.
       * @see CapSense
       */
      public CapSense openCapSense(int pin, float filterCoef) throws
ConnectionLostException;

      /**
       * Open a motion-control sequencer.
       * <p>
       * This module allows fast and precise sequencing of waveforms,
primarily intended for
       * synchronized driving of various kinds of motors and other actuators.
There is currently
       * support for only a single instance of this module. For more details,
see {@link Sequencer}.
       *
       * @param config
       *            The sequencer configuration.
       * @return The sequencer instance.
       * @throws ConnectionLostException
       *             Connection was lost before or during the execution of
this method.
       */
      public Sequencer openSequencer(Sequencer.ChannelConfig config[]) throws
ConnectionLostException;

      /**
       * Start a batch of operations. This is strictly an optimization and
will not change
       * functionality: if the client knows that a sequence of several IOIO
operations are going to be
       * performed immediately following each other, a call to {@link
#beginBatch()} before the
       * sequence and {@link #endBatch()} after the sequence will cause the
operations to be grouped
       * into one transfer to the IOIO, thus reducing latency. A matching
{@link #endBatch()}
       * operation must always follow, or otherwise no operation will ever be
actually executed.
       * {@link #beginBatch()} / {@link #endBatch()} blocks may be nested -
the transfer will occur
       * when the outermost {@link #endBatch()} is invoked. Note that it is
not guaranteed that no
       * transfers will happen while inside a batch - it should be treated as
a hint. Code running
       * inside the block must be quick as it blocks <b>all</b> transfers to
the IOIO, including those
       * performed from other threads.
       *
       * @throws ConnectionLostException
```

```
         *             Connection was lost before or during the execution of
this method.
         */
        public void beginBatch() throws ConnectionLostException;

        /**
         * End a batch of operations. For explanation, see {@link
#beginBatch()}.
         *
         * @throws ConnectionLostException
         *             Connection was lost before or during the execution of
this method.
         */
        public void endBatch() throws ConnectionLostException;

        /**
         * Sends a message to the IOIO and waits for an echo.
         *
         * This is useful for synchronizing asynchronous calls across the
entire API, for example: When
         * writing to a {@link DigitalOutput} and then reading from a {@link
DigitalInput}, if you want
         * to guarantee that the reading was obtained after the write has taken
place, call this method
         * in between.
         *
         * @throws ConnectionLostException
         *             Connection was lost before or during the execution of
this method.
         * @throws InterruptedException
         *             When interrupt() has been called on this thread.
         */
        public void sync() throws ConnectionLostException,
InterruptedException;

        /**
         * Returns the result of an Analogic-Digital Conversion.
         * @return ADCResult
         * @throws ConnectionLostException
         */
        public int[] getADCResult() throws ConnectionLostException;

        /**
         * Returns the number of messages remaining to complete the transfer of
the ADC result.
         * @return numMsg
         * @throws ConnectionLostException
         */
        public int getNumMsg() throws ConnectionLostException;

        /**
         * Sends a message to IOIO in order to start a new offline ADC.
         * @throws ConnectionLostException
         */
        public void setAnalogOffline() throws ConnectionLostException;

        /**
         * Sends a message to IOIO in order to stop an offline ADC and start to
transfer the result.
         * @throws ConnectionLostException
         */
        public void setReturnAnalogOfflineStatus() throws
```

```
ConnectionLostException;

}
```

IOIOImpl.java

```
package ioio.lib.impl;

import ioio.lib.api.AnalogInput;
import ioio.lib.api.CapSense;
import ioio.lib.api.DigitalInput;
import ioio.lib.api.DigitalInput.Spec;
import ioio.lib.api.DigitalInput.Spec.Mode;
import ioio.lib.api.DigitalOutput;
import ioio.lib.api.IOIO;
import ioio.lib.api.IOIOConnection;
import ioio.lib.api.IcspMaster;
import ioio.lib.api.PulseInput;
import ioio.lib.api.PulseInput.ClockRate;
import ioio.lib.api.PulseInput.PulseMode;
import ioio.lib.api.PwmOutput;
import ioio.lib.api.Sequencer;
import ioio.lib.api.SpiMaster;
import ioio.lib.api.TwiMaster;
import ioio.lib.api.TwiMaster.Rate;
import ioio.lib.api.Uart;
import ioio.lib.api.exception.ConnectionLostException;
import ioio.lib.api.exception.IncompatibilityException;
import ioio.lib.impl.IOIOProtocol.PwmScale;
import ioio.lib.impl.IncomingState.DisconnectListener;
import ioio.lib.impl.ResourceManager.Resource;
import ioio.lib.impl.ResourceManager.ResourceType;
import ioio.lib.spi.Log;

import java.io.IOException;

public class IOIOImpl implements IOIO, DisconnectListener {
      private static class SyncListener implements
IncomingState.SyncListener, DisconnectListener {
            enum State { WAITING, SIGNALED, DISCONNECTED };
            private State state_ = State.WAITING;

            @Override
            public synchronized void sync() {
                  state_ = State.SIGNALED;
                  notifyAll();
            }

            public synchronized void waitSync() throws InterruptedException,
ConnectionLostException {
                  while (state_ == State.WAITING) {
                        wait();
                  }
                  if (state_ == State.DISCONNECTED) {
                        throw new ConnectionLostException();
                  }
            }

            @Override
```

```
            public synchronized void disconnected() {
                  state_ = State.DISCONNECTED;
                  notifyAll();
            }
      }

      private static final String TAG = "IOIOImpl";
      private boolean disconnect_ = false;

      private static final byte[] REQUIRED_INTERFACE_ID = new byte[] { 'I',
'O',
                  'I', 'O', '0', '0', '0', '5' };

      IOIOProtocol protocol_;
      ResourceManager resourceManager_;
      IncomingState incomingState_ = new IncomingState();
      Board.Hardware hardware_;
      private IOIOConnection connection_;
      private State state_ = State.INIT;

      public IOIOImpl(IOIOConnection con) {
            connection_ = con;
      }

      @Override
      public void waitForConnect() throws ConnectionLostException,
                  IncompatibilityException {
            if (state_ == State.CONNECTED) {
                  return;
            }
            if (state_ == State.DEAD) {
                  throw new ConnectionLostException();
            }
            addDisconnectListener(this);
            Log.d(TAG, "Waiting for IOIO connection");
            try {
                  try {
                        Log.v(TAG, "Waiting for underlying connection");
                        connection_.waitForConnect();
                        synchronized (this) {
                              if (disconnect_) {
                                    throw new ConnectionLostException();
                              }
                              protocol_ = new
IOIOProtocol(connection_.getInputStream(),
                                          connection_.getOutputStream(),
incomingState_);
                              // Once this block exits, a disconnect will also
involve
                              // softClose().
                        }
                  } catch (ConnectionLostException e) {
                        incomingState_.handleConnectionLost();
                        throw e;
                  }
                  Log.v(TAG, "Waiting for handshake");
                  incomingState_.waitConnectionEstablished();
                  initBoard();
                  Log.v(TAG, "Querying for required interface ID");
                  checkInterfaceVersion();
                  Log.v(TAG, "Required interface ID is supported");
                  state_ = State.CONNECTED;
```

```
                Log.i(TAG, "IOIO connection established");
            } catch (ConnectionLostException e) {
                Log.d(TAG, "Connection lost / aborted");
                state_ = State.DEAD;
                throw e;
            } catch (IncompatibilityException e) {
                throw e;
            } catch (InterruptedException e) {
                Log.e(TAG, "Unexpected exception", e);
            }
        }

        @Override
        public synchronized void disconnect() {
            Log.d(TAG, "Client requested disconnect.");
            if (disconnect_) {
                return;
            }
            disconnect_ = true;
            try {
                if (protocol_ != null && !connection_.canClose()) {
                    protocol_.softClose();
                }
            } catch (IOException e) {
                Log.e(TAG, "Soft close failed", e);
            }
            connection_.disconnect();
        }

        @Override
        public synchronized void disconnected() {
            state_ = State.DEAD;
            if (disconnect_) {
                return;
            }
            Log.d(TAG, "Physical disconnect.");
            disconnect_ = true;
            // The IOIOConnection doesn't necessarily know about the
disconnect
            connection_.disconnect();
        }

        @Override
        public void waitForDisconnect() throws InterruptedException {
            incomingState_.waitDisconnect();
        }

        @Override
        public State getState() {
            return state_;
        }

        private void initBoard() throws IncompatibilityException {
            if (incomingState_.board_ == null) {
                throw new IncompatibilityException("Unknown board: "
                        + incomingState_.hardwareId_);
            }
            hardware_ = incomingState_.board_.hardware;
            resourceManager_ = new ResourceManager(hardware_);
        }

        private void checkInterfaceVersion() throws IncompatibilityException,
```

```
                  ConnectionLostException, InterruptedException {
            try {
                  protocol_.checkInterface(REQUIRED_INTERFACE_ID);
            } catch (IOException e) {
                  throw new ConnectionLostException(e);
            }
            if (!incomingState_.waitForInterfaceSupport()) {
                  state_ = State.INCOMPATIBLE;
                  Log.e(TAG, "Required interface ID is not supported");
                  throw new IncompatibilityException(
                              "IOIO firmware does not support required
firmware: "
                                          + new
String(REQUIRED_INTERFACE_ID));
            }
      }

      synchronized void removeDisconnectListener(DisconnectListener listener)
{
            incomingState_.removeDisconnectListener(listener);
      }

      synchronized void addDisconnectListener(DisconnectListener listener)
                  throws ConnectionLostException {
            incomingState_.addDisconnectListener(listener);
      }

      synchronized void closePin(ResourceManager.Resource pin) {
            try {
                  protocol_.setPinDigitalIn(pin.id,
DigitalInput.Spec.Mode.FLOATING);
                  resourceManager_.free(pin);
            } catch (IOException e) {
            }
      }

      @Override
      synchronized public void softReset() throws ConnectionLostException {
            checkState();
            try {
                  protocol_.softReset();
            } catch (IOException e) {
                  throw new ConnectionLostException(e);
            }
      }

      @Override
      synchronized public void hardReset() throws ConnectionLostException {
            checkState();
            try {
                  protocol_.hardReset();
            } catch (IOException e) {
                  throw new ConnectionLostException(e);
            }
      }

      @Override
      public String getImplVersion(VersionType v) {
            if (state_ == State.INIT) {
                  throw new IllegalStateException(
                              "Connection has not yet been established");
            }
```

```
            switch (v) {
            case HARDWARE_VER:
                  return incomingState_.hardwareId_;
            case BOOTLOADER_VER:
                  return incomingState_.bootloaderId_;
            case APP_FIRMWARE_VER:
                  return incomingState_.firmwareId_;
            case IOIOLIB_VER:
                  return "IOIO0504";
            }
            return null;
      }

      @Override
      public DigitalInput openDigitalInput(int pin)
                  throws ConnectionLostException {
            return openDigitalInput(new DigitalInput.Spec(pin));
      }

      @Override
      public DigitalInput openDigitalInput(int pin, Mode mode)
                  throws ConnectionLostException {
            return openDigitalInput(new DigitalInput.Spec(pin, mode));
      }

      @Override
      synchronized public DigitalInput openDigitalInput(DigitalInput.Spec
spec)
                  throws ConnectionLostException {
            checkState();
            Resource pin = new Resource(ResourceType.PIN, spec.pin);
            resourceManager_.alloc(pin);
            DigitalInputImpl result = new DigitalInputImpl(this, pin);
            addDisconnectListener(result);
            incomingState_.addInputPinListener(spec.pin, result);
            try {
                  protocol_.setPinDigitalIn(spec.pin, spec.mode);
                  protocol_.setChangeNotify(spec.pin, true);
            } catch (IOException e) {
                  result.close();
                  throw new ConnectionLostException(e);
            }
            return result;
      }

      @Override
      public DigitalOutput openDigitalOutput(int pin,
                  ioio.lib.api.DigitalOutput.Spec.Mode mode, boolean
startValue)
                  throws ConnectionLostException {
            return openDigitalOutput(new DigitalOutput.Spec(pin, mode),
startValue);
      }

      @Override
      synchronized public DigitalOutput openDigitalOutput(
                  DigitalOutput.Spec spec, boolean startValue)
                  throws ConnectionLostException {
            checkState();
            Resource pin = new Resource(ResourceType.PIN, spec.pin);
            resourceManager_.alloc(pin);
```

```
            DigitalOutputImpl result = new DigitalOutputImpl(this, pin,
startValue);
            addDisconnectListener(result);
            try {
                  protocol_.setPinDigitalOut(spec.pin, startValue, spec.mode);
            } catch (IOException e) {
                  result.close();
                  throw new ConnectionLostException(e);
            }
            return result;
      }

      @Override
      public DigitalOutput openDigitalOutput(int pin, boolean startValue)
                  throws ConnectionLostException {
            return openDigitalOutput(new DigitalOutput.Spec(pin), startValue);
      }

      @Override
      public DigitalOutput openDigitalOutput(int pin)
                  throws ConnectionLostException {
            return openDigitalOutput(new DigitalOutput.Spec(pin), false);
      }

      @Override
      synchronized public AnalogInput openAnalogInput(int pinNum)
                  throws ConnectionLostException {
            checkState();
            hardware_.checkSupportsAnalogInput(pinNum);
            Resource pin = new Resource(ResourceType.PIN, pinNum);
            resourceManager_.alloc(pin);
            AnalogInputImpl result = new AnalogInputImpl(this, pin);
            addDisconnectListener(result);
            incomingState_.addInputPinListener(pinNum, result);
            try {
                  protocol_.setPinAnalogIn(pinNum);
                  protocol_.setAnalogInSampling(pinNum, true);
            } catch (IOException e) {
                  result.close();
                  throw new ConnectionLostException(e);
            }
            return result;
      }

      @Override
      public CapSense openCapSense(int pin) throws ConnectionLostException {
            return openCapSense(pin, CapSense.DEFAULT_COEF);
      }

      @Override
      public synchronized CapSense openCapSense(int pinNum, float filterCoef)
                  throws ConnectionLostException {
            checkState();
            hardware_.checkSupportsCapSense(pinNum);
            Resource pin = new Resource(ResourceType.PIN, pinNum);
            resourceManager_.alloc(pin);
            CapSenseImpl result = new CapSenseImpl(this, pin, filterCoef);
            addDisconnectListener(result);
            incomingState_.addInputPinListener(pinNum, result);
            try {
                  protocol_.setPinCapSense(pinNum);
                  protocol_.setCapSenseSampling(pinNum, true);
```

```
            } catch (IOException e) {
                  result.close();
                  throw new ConnectionLostException(e);
            }
            return result;
      }

      @Override
      public PwmOutput openPwmOutput(int pin, int freqHz)
                  throws ConnectionLostException {
            return openPwmOutput(new DigitalOutput.Spec(pin), freqHz);
      }

      @Override
      synchronized public PwmOutput openPwmOutput(DigitalOutput.Spec spec,
                  int freqHz) throws ConnectionLostException {
            checkState();
            hardware_.checkSupportsPeripheralOutput(spec.pin);

            Resource pin = new Resource(ResourceType.PIN, spec.pin);
            Resource oc = new Resource(ResourceType.OUTCOMPARE);

            resourceManager_.alloc(pin, oc);
            int scale = 0;
            float baseUs;
            int period;
            while (true) {
                  final int clk = 16000000 /
IOIOProtocol.PwmScale.values()[scale].scale;
                  period = clk / freqHz;
                  if (period <= 65536) {
                        baseUs = 1000000.0f / clk;
                        break;
                  }
                  if (++scale >= PwmScale.values().length) {
                        throw new IllegalArgumentException("Frequency too low:
"
                                    + freqHz);
                  }
            }

            PwmImpl pwm = new PwmImpl(this, pin, oc, period, baseUs);
            addDisconnectListener(pwm);
            try {
                  protocol_.setPinDigitalOut(spec.pin, false, spec.mode);
                  protocol_.setPinPwm(spec.pin, oc.id, true);
                  protocol_.setPwmPeriod(oc.id, period - 1,
                              IOIOProtocol.PwmScale.values()[scale]);
            } catch (IOException e) {
                  pwm.close();
                  throw new ConnectionLostException(e);
            }
            return pwm;
      }

      @Override
      public Uart openUart(int rx, int tx, int baud, Uart.Parity parity,
                  Uart.StopBits stopbits) throws ConnectionLostException {
            return openUart(rx == INVALID_PIN ? null : new
DigitalInput.Spec(rx),
                        tx == INVALID_PIN ? null : new DigitalOutput.Spec(tx),
baud,
```

```
                        parity, stopbits);
      }

      @Override
      synchronized public Uart openUart(DigitalInput.Spec rx,
                  DigitalOutput.Spec tx, int baud, Uart.Parity parity,
                  Uart.StopBits stopbits) throws ConnectionLostException {
            checkState();
            if (rx != null) {
                  hardware_.checkSupportsPeripheralInput(rx.pin);
            }
            if (tx != null) {
                  hardware_.checkSupportsPeripheralOutput(tx.pin);
            }
            Resource rxPin = rx != null ? new Resource(ResourceType.PIN,
rx.pin)
                        : null;
            Resource txPin = tx != null ? new Resource(ResourceType.PIN,
tx.pin)
                        : null;
            Resource uart = new Resource(ResourceType.UART);
            resourceManager_.alloc(rxPin, txPin, uart);

            UartImpl result = new UartImpl(this, txPin, rxPin, uart);
            addDisconnectListener(result);
            incomingState_.addUartListener(uart.id, result);
            try {
                  if (rx != null) {
                        protocol_.setPinDigitalIn(rx.pin, rx.mode);
                        protocol_.setPinUart(rx.pin, uart.id, false, true);
                  }
                  if (tx != null) {
                        protocol_.setPinDigitalOut(tx.pin, true, tx.mode);
                        protocol_.setPinUart(tx.pin, uart.id, true, true);
                  }
                  boolean speed4x = true;
                  int rate = Math.round(4000000.0f / baud) - 1;
                  if (rate > 65535) {
                        speed4x = false;
                        rate = Math.round(1000000.0f / baud) - 1;
                  }
                  protocol_.uartConfigure(uart.id, rate, speed4x, stopbits,
parity);
            } catch (IOException e) {
                  result.close();
                  throw new ConnectionLostException(e);
            }
            return result;
      }

      @Override
      synchronized public TwiMaster openTwiMaster(int twiNum, Rate rate,
                  boolean smbus) throws ConnectionLostException {
            checkState();

            final int[][] twiPins = hardware_.twiPins();

            Resource twi = new Resource(ResourceType.TWI, twiNum);
            Resource[] pins = new Resource[] {
                        new Resource(ResourceType.PIN, twiPins[twiNum][0]),
                        new Resource(ResourceType.PIN, twiPins[twiNum][1]) };
```

```
            resourceManager_.alloc(twi, pins);

            TwiMasterImpl result = new TwiMasterImpl(this, twi, pins);
            addDisconnectListener(result);
            incomingState_.addTwiListener(twiNum, result);
            try {
                  protocol_.i2cConfigureMaster(twiNum, rate, smbus);
            } catch (IOException e) {
                  result.close();
                  throw new ConnectionLostException(e);
            }
            return result;
      }

      @Override
      synchronized public IcspMaster openIcspMaster()
                  throws ConnectionLostException {
            checkState();

            final int[] icspPins = hardware_.icspPins();
            Resource icsp = new Resource(ResourceType.ICSP);
            Resource[] pins = new Resource[] {
                        new Resource(ResourceType.PIN, icspPins[0]),
                        new Resource(ResourceType.PIN, icspPins[1]),
                        new Resource(ResourceType.PIN, icspPins[2]) };

            resourceManager_.alloc(icsp, pins);

            IcspMasterImpl result = new IcspMasterImpl(this, icsp, pins);
            addDisconnectListener(result);
            incomingState_.addIcspListener(result);
            try {
                  protocol_.icspOpen();
            } catch (IOException e) {
                  result.close();
                  throw new ConnectionLostException(e);
            }
            return result;
      }

      @Override
      public SpiMaster openSpiMaster(int miso, int mosi, int clk,
                  int slaveSelect, SpiMaster.Rate rate)
                  throws ConnectionLostException {
            return openSpiMaster(miso, mosi, clk, new int[] { slaveSelect },
rate);
      }

      @Override
      public SpiMaster openSpiMaster(int miso, int mosi, int clk,
                  int[] slaveSelect, SpiMaster.Rate rate)
                  throws ConnectionLostException {
            DigitalOutput.Spec[] slaveSpecs = new
DigitalOutput.Spec[slaveSelect.length];
            for (int i = 0; i < slaveSelect.length; ++i) {
                  slaveSpecs[i] = new DigitalOutput.Spec(slaveSelect[i]);
            }
            return openSpiMaster(new DigitalInput.Spec(miso, Mode.PULL_UP),
                        new DigitalOutput.Spec(mosi), new
DigitalOutput.Spec(clk),
                        slaveSpecs, new SpiMaster.Config(rate));
      }
```

```
	@Override
	synchronized public SpiMaster openSpiMaster(DigitalInput.Spec miso,
			DigitalOutput.Spec mosi, DigitalOutput.Spec clk,
			DigitalOutput.Spec[] slaveSelect, SpiMaster.Config config)
			throws ConnectionLostException {
		checkState();

		hardware_.checkSupportsPeripheralInput(miso.pin);
		hardware_.checkSupportsPeripheralOutput(mosi.pin);
		hardware_.checkSupportsPeripheralOutput(clk.pin);

		Resource ssPins[] = new Resource[slaveSelect.length];
		Resource misoPin = new Resource(ResourceType.PIN, miso.pin);
		Resource mosiPin = new Resource(ResourceType.PIN, mosi.pin);
		Resource clkPin = new Resource(ResourceType.PIN, clk.pin);
		for (int i = 0; i < slaveSelect.length; ++i) {
			ssPins[i] = new Resource(ResourceType.PIN,
slaveSelect[i].pin);
		}
		Resource spi = new Resource(ResourceType.SPI);

		resourceManager_.alloc(ssPins, misoPin, mosiPin, clkPin, spi);

		SpiMasterImpl result = new SpiMasterImpl(this, spi, mosiPin,
misoPin, clkPin, ssPins);
		addDisconnectListener(result);

		incomingState_.addSpiListener(spi.id, result);
		try {
			protocol_.setPinDigitalIn(miso.pin, miso.mode);
			protocol_.setPinSpi(miso.pin, 1, true, spi.id);
			protocol_.setPinDigitalOut(mosi.pin, true, mosi.mode);
			protocol_.setPinSpi(mosi.pin, 0, true, spi.id);
			protocol_.setPinDigitalOut(clk.pin, config.invertClk,
clk.mode);
			protocol_.setPinSpi(clk.pin, 2, true, spi.id);
			for (DigitalOutput.Spec spec : slaveSelect) {
				protocol_.setPinDigitalOut(spec.pin, true, spec.mode);
			}
			protocol_.spiConfigureMaster(spi.id, config);
		} catch (IOException e) {
			result.close();
			throw new ConnectionLostException(e);
		}
		return result;
	}

	@Override
	public PulseInput openPulseInput(Spec spec, ClockRate rate, PulseMode
mode,
			boolean doublePrecision) throws ConnectionLostException {
		checkState();
		hardware_.checkSupportsPeripheralInput(spec.pin);
		Resource pin = new Resource(ResourceType.PIN, spec.pin);
		Resource incap = new Resource(
				doublePrecision ? ResourceType.INCAP_DOUBLE
						: ResourceType.INCAP_SINGLE);
		resourceManager_.alloc(pin, incap);

		IncapImpl result = new IncapImpl(this, mode, incap, pin,
rate.hertz,
```

```
                        mode.scaling, doublePrecision);
            addDisconnectListener(result);
            incomingState_.addIncapListener(incap.id, result);
            try {
                  protocol_.setPinDigitalIn(spec.pin, spec.mode);
                  protocol_.setPinIncap(spec.pin, incap.id, true);
                  protocol_.incapConfigure(incap.id, doublePrecision,
                              mode.ordinal() + 1, rate.ordinal());
            } catch (IOException e) {
                  result.close();
                  throw new ConnectionLostException(e);
            }
            return result;
      }

      @Override
      public PulseInput openPulseInput(int pin, PulseMode mode)
                  throws ConnectionLostException {
            return openPulseInput(new DigitalInput.Spec(pin),
ClockRate.RATE_16MHz,
                        mode, true);
      }

      @Override
      public Sequencer openSequencer(Sequencer.ChannelConfig config[])
                  throws ConnectionLostException {
            return new SequencerImpl(this, config);
      }

      private void checkState() throws ConnectionLostException {
            if (state_ == State.DEAD) {
                  throw new ConnectionLostException();
            }
            if (state_ == State.INCOMPATIBLE) {
                  throw new IllegalStateException(
                              "Incompatibility has been reported - IOIO cannot
be used");
            }
            if (state_ != State.CONNECTED) {
                  throw new IllegalStateException(
                              "Connection has not yet been established");
            }
      }

      @Override
      public synchronized void beginBatch() throws ConnectionLostException {
            checkState();
            protocol_.beginBatch();
      }

      @Override
      public synchronized void endBatch() throws ConnectionLostException {
            checkState();
            try {
                  protocol_.endBatch();
            } catch (IOException e) {
                  throw new ConnectionLostException(e);
            }
      }

      @Override
      public void sync() throws ConnectionLostException, InterruptedException
```

```
{
            boolean added = false;
            SyncListener listener = new SyncListener();
            try {
                  synchronized (this) {
                        checkState();
                        incomingState_.addSyncListener(listener);
                        addDisconnectListener(listener);
                        added = true;
                        try {
                              protocol_.sync();
                        } catch (IOException e) {
                              throw new ConnectionLostException(e);
                        }
                  }
                  listener.waitSync();
            } finally {
                  if (added) {
                        removeDisconnectListener(listener);
                  }
            }
      }

      @Override
      public int[] getADCResult() throws ConnectionLostException {
            try{
                  Log.v(TAG, "Muestras recibidas (IOIOImpl): "+
incomingState_.ADCResult_);
                  return incomingState_.ADCResult_;
            } finally {

            }
      }

      @Override
      public int getNumMsg() throws ConnectionLostException {
            try{
                  Log.v(TAG, "Mensajes restantes (IOIOImpl): "+
incomingState_.numMsg_);
                  return incomingState_.numMsg_;
            } finally {

            }
      }

      @Override
      public void setAnalogOffline() throws ConnectionLostException {
            try {
                  protocol_.setAnalogOffline();
            } catch (IOException e) {
                  throw new ConnectionLostException(e);
            }
      }

      @Override
      public void setReturnAnalogOfflineStatus() throws
ConnectionLostException {
            try {
                  protocol_.setReturnAnalogOfflineStatus();
            } catch (IOException e) {
                  throw new ConnectionLostException(e);
            }
```

```
        }
}
```

3. Aplicación

MainActivity.java

```
package com.dinel.javi.signalmonitoring;

import android.app.AlertDialog;
import android.content.DialogInterface;
import android.content.Intent;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.View;
import android.widget.ImageButton;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Calendar;
import java.util.Date;
import java.util.LinkedList;
import java.util.List;

import java.io.*;

import ioio.lib.api.IOIO;
import ioio.lib.api.exception.ConnectionLostException;
import ioio.lib.util.BaseIOIOLooper;
import ioio.lib.util.IOIOLooper;
import ioio.lib.util.android.IOIOActivity;


public class MainActivity extends IOIOActivity {

    private ImageButton setADCButton;
    private ImageButton returnADCButton;
    private ImageButton seeChartButton;

    private IOIO myioio;
    private List<int[]> msgResultList = new LinkedList<>();
    private int[] ADCResult;
    private Date dateSet;
    public static final String TAG = "IOIOAPP";


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        setADCButton = (ImageButton) findViewById(R.id.btnSetADC);
        returnADCButton = (ImageButton) findViewById(R.id.btnReturnADC);
        seeChartButton = (ImageButton) findViewById(R.id.btnChart);

        setADCButton.setEnabled(false);
        returnADCButton.setEnabled(false);
        seeChartButton.setEnabled(false);
    }

      /**
       * Send SET_ADC_OFFLINE message to IOIO
       */
    public void setADCOffline(View view) throws ConnectionLostException,
```

```
InterruptedException {
        myioio.setAnalogOffline();
        myioio.disconnect();
        dateSet = Calendar.getInstance().getTime();
    }

    /**
       * Recover ADC results from IOIO
       */
    public void returnADCResult(View view) throws ConnectionLostException,
InterruptedException {
        int[] msgResult;
        int numMsg = 0;
            // Send SET_RETURN_ANALOG_OFFLINE messages until there is one
result message left
        while (numMsg != 1) {
            myioio.setReturnAnalogOfflineStatus();
            Thread.sleep(300);
            numMsg = myioio.getNumMsg();
            Thread.sleep(300);
            Log.v(TAG, "numMsg= " + numMsg);
            msgResult = myioio.getADCResult();
            Thread.sleep(300);
            Log.v(TAG, "ADCResult" + numMsg + "= " +
Arrays.toString(msgResult));
            msgResultList.add(myioio.getADCResult());
        }
        int arrayLength = 0;
        for (int i = 0; i < msgResultList.size(); i++) {
            msgResult = msgResultList.get(i);
            arrayLength += msgResult.length;
        }
            // Array with all the samples
        ADCResult = new int[arrayLength];
        int samples = 0;
        for (int i = 0; i < msgResultList.size(); i++) {
            msgResult = msgResultList.get(i);
            for (int j = 0; j < msgResult.length && samples <
ADCResult.length; j++) {
                ADCResult[samples] = msgResult[j];
                samples++;
            }
        }
        Log.v(TAG, "ADCResult=" + Arrays.toString(ADCResult));
            // Show alert
        alertResult();
        myioio.disconnect();
    }

      /**
       * Go to activity SeeChartActivity
       */
    public void seeChart(View view) throws ConnectionLostException,
InterruptedException {
        Intent intent = new Intent(this, SeeChartActivity.class);
        intent.putExtra("ADCResult", ADCResult);
        startActivity(intent);
    }

    /**
       * Save the samples in a CSV file
       */
```

```
    public void saveCSV(View view) throws ConnectionLostException,
InterruptedException {
        File folder = new File(Environment.getExternalStorageDirectory() +
                File.separator + "IOIO");
        boolean success = true;
        if (!folder.exists()) { // if folder doesn't exist, create it
            success = folder.mkdirs();
        }
        if (success) { // Create file
            SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddHHmmss");
            File file = new File(Environment.getExternalStorageDirectory() +
                    File.separator + "IOIO" + File.separator + "Samples" +
sdf.format(dateSet) + ".csv");
            if (!file.exists()) {
                try {
                    file.createNewFile();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                try {
                    FileOutputStream fos = new FileOutputStream(file);

                    String sampleToWrite;
                    for (int i=0; i<ADCResult.length; i++) {
                        sampleToWrite = Integer.toString(ADCResult[i]);
                        if ((i + 1) < ADCResult.length) {
                            sampleToWrite += ",";
                        }
                        fos.write(sampleToWrite.getBytes());
                    }
                    alertCSV(file.getPath());
                    fos.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        } else { //Failure
            // Do something else on failure
        }
    }

    class Looper extends BaseIOIOLooper {

        @Override
        protected void setup() throws ConnectionLostException {
            myioio = ioio_;
            enableUi(true);
        }

        @Override
        public void loop() throws ConnectionLostException {

        }

        @Override
        public void disconnected() {
            enableUi(false);
        }
    }

    @Override
    protected IOIOLooper createIOIOLooper() {
```

```
        return new Looper();
    }

    private int numConnected_ = 0;
    private void enableUi(final boolean enable) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                if (enable) {
                    if (numConnected_++ == 0) {
                        setADCButton.setEnabled(true);
                        returnADCButton.setEnabled(true);
                        seeChartButton.setEnabled(true);
                    }
                } else {
                    if (--numConnected_ == 0) {
                        setADCButton.setEnabled(false);
                        returnADCButton.setEnabled(false);
                        seeChartButton.setEnabled(false);
                    }
                }
            }
        });
    }

    private void alertResult() {
        AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
        dialogBuilder.setMessage("Samples have been obtained.");
        dialogBuilder.setCancelable(true).setTitle("Data received");

        dialogBuilder.setPositiveButton("OK", new
DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                dialog.cancel();
            }
        });
        dialogBuilder.create().show();
    }

    private void alertCSV(String path) {
        AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
        dialogBuilder.setMessage("Samples have been saved to " + path);
        dialogBuilder.setCancelable(true).setTitle("Data saved");

        dialogBuilder.setPositiveButton("OK", new
DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                dialog.cancel();
            }
        });
        dialogBuilder.create().show();
    }

}
```

SeeChartActivity.java

```
package com.dinel.javi.signalmonitoring;

import android.app.Activity;
import android.graphics.Color;
import android.os.Bundle;
import android.util.Log;

import com.androidplot.xy.LineAndPointFormatter;
import com.androidplot.xy.SimpleXYSeries;
import com.androidplot.xy.XYPlot;
import com.androidplot.xy.XYSeries;

import java.util.Arrays;

public class SeeChartActivity extends Activity {

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

        // initialize our XYPlot reference:
        XYPlot plot;
        plot = (XYPlot) findViewById(R.id.plot);

        int[] ADCResult;
        ADCResult=getIntent().getExtras().getIntArray("ADCResult");
        float [] VoltageADC = new float[ADCResult.length];
        for (int i=0; i<ADCResult.length; i++) {
            VoltageADC[i]=ADCResult[i];
        }
        Number[] seriesNumbers=new Number[ADCResult.length];
        for(int i=0; i<ADCResult.length; i++) {
            seriesNumbers[i]=VoltageADC[i]/1023*3.3;
            Log.v(MainActivity.TAG, Arrays.toString(seriesNumbers));
        }

        // turn the above arrays into XYSeries':
        // (Y_VALS_ONLY means use the element index as the x value)
        XYSeries series = new SimpleXYSeries(Arrays.asList(seriesNumbers),
SimpleXYSeries.ArrayFormat.Y_VALS_ONLY, "Series");
        LineAndPointFormatter seriesFormat = new
LineAndPointFormatter(Color.rgb(127, 255, 0), 0x000000, 0x000000, null);
        //setText(Arrays.toString(ADCResult));
        plot.addSeries(series, seriesFormat);
    }

}
```

ANEXO C: INSTRUCCIONES PARA CONFIGURAR EL ENTORNO DE DESARROLLO

Para comenzar a producir con IOIO y Android es necesario disponer de los diferentes entornos de desarrollo mencionados en el capítulo 2 de este documento. En este anexo se describirá detalladamente cómo instalar y configurar cada uno de los entornos y los proyectos necesarios para modificar y hacer funcionar IOIO y sus aplicaciones en Android. Todos estos pasos serán detallados para un S.O. basado en Linux, aunque las únicas diferencias respecto a otros sistemas operativos solo se encontrarán en la instalación de los entornos de desarrollo, ya que la configuración, una vez instalados, será similar.

1. MPLAB X

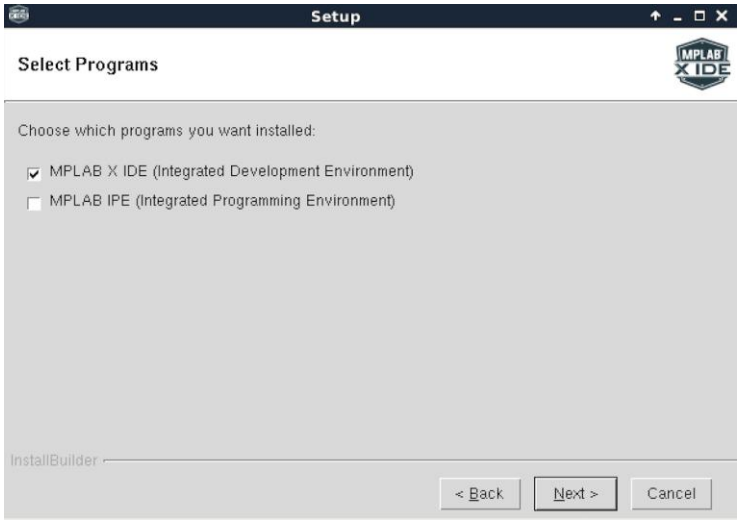
En primer lugar, se comenzará con la instalación de MPLAB X, el entorno de desarrollo necesario para programar el microcontrolador PIC de IOIO. La versión a descargar será la v3.5.0, disponible para cualquier SO desde http://www.microchip.com/development-tools/downloads-archive. En este mismo sitio web, será necesario descargar el compilador XC16, indispensable para compilar el código fuente del firmware de IOIO. La versión a descargar será la v1.24.

Una vez descargados, se comenzará con la instalación. En Linux, el instalador del IDE es un script que habrá que ejecutar como superusuario desde un terminal:

```
root@debian:/opt# ./MPLABX-v3.50-linux-installer.sh
64 Bit, check libraries
Check for 32 Bit libraries
Verifying archive integrity... All good.
Uncompressing MPLAB X v3.50 Installer....
```

MPLAB X y el compilador son aplicaciones de 32 bits, por lo que en caso de estar utilizando una distribución Linux de 64 bits puede ser necesaria la instalación de paquetes adicionales. En el siguiente enlace se muestra una lista de los paquetes que podrían ser necesarios instalar en ese caso http://microchipdeveloper.com/install:mplabx-lin64.

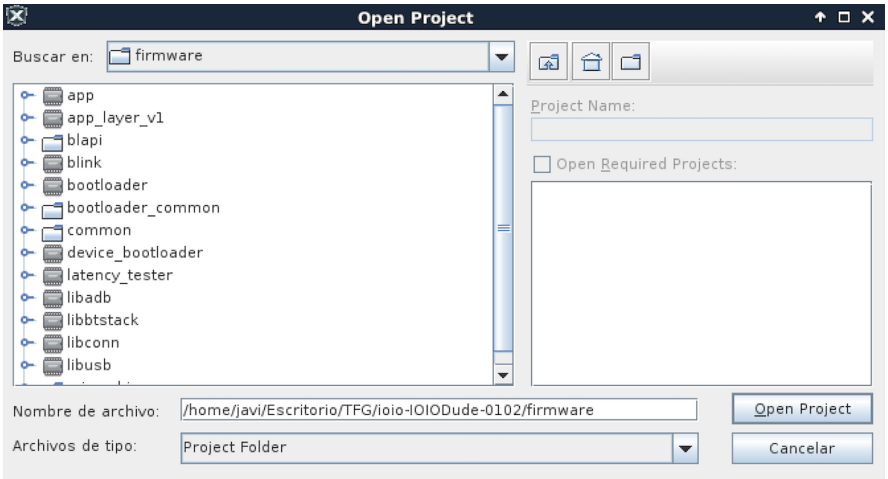
Tras esto, se abrirá una nueva ventana para la instalación del programa. En uno de los pasos habrá que indicar los programas a instalar, donde habrá que desmarcar la opción del IPE, ya que para el propósito de este proyecto no es necesario:



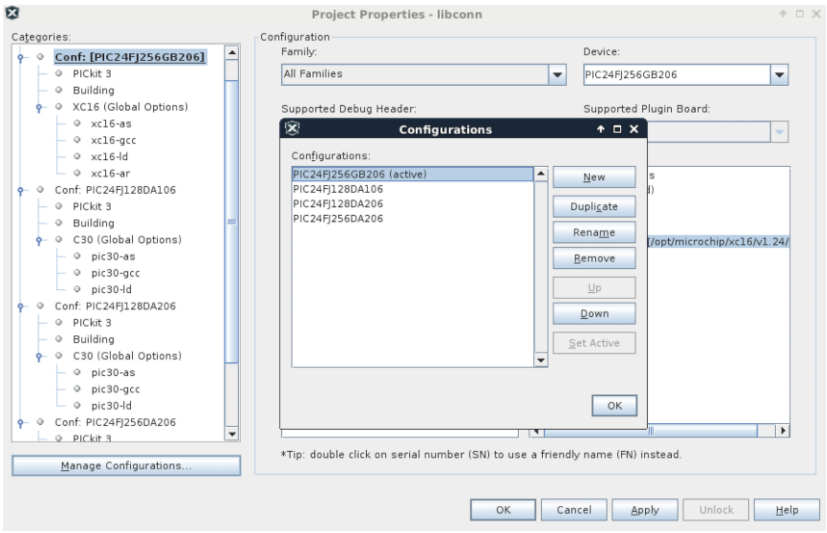
El siguiente paso a realizar, una vez terminada la instalación de MPLAB, es instalar el compilador XC16. Será necesario ejecutar el archivo descargado, no sin antes darle permisos de ejecución:

```
root@debian:/opt# chmod +x xc16-v1.24-full-install-linux-installer.run
root@debian:/opt# ./xc16-v1.24-full-install-linux-installer.run
```

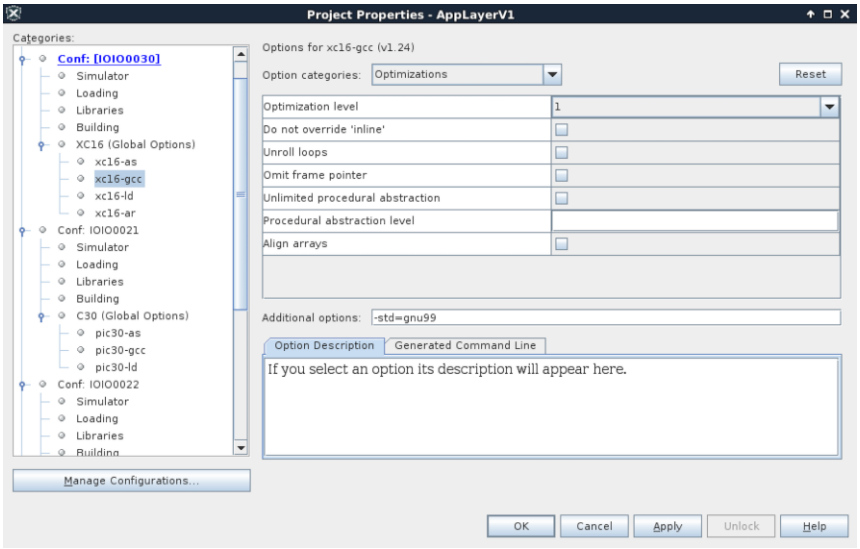
Al igual que antes, se abrirá una ventana para la instalación del compilador, donde siguiendo todos los pasos se terminará la instalación con éxito. A partir de este momento ya será posible incluir todos los proyectos para desarrollar IOIO. Estos se pueden obtener en https://github.com/ytai/ioio, en el directorio "firmware". Así, en MPLAB habrá que ir a File>Open Project… y seleccionar los proyectos libusb, libconn, libbtstack, libadb y app_layer_v1:



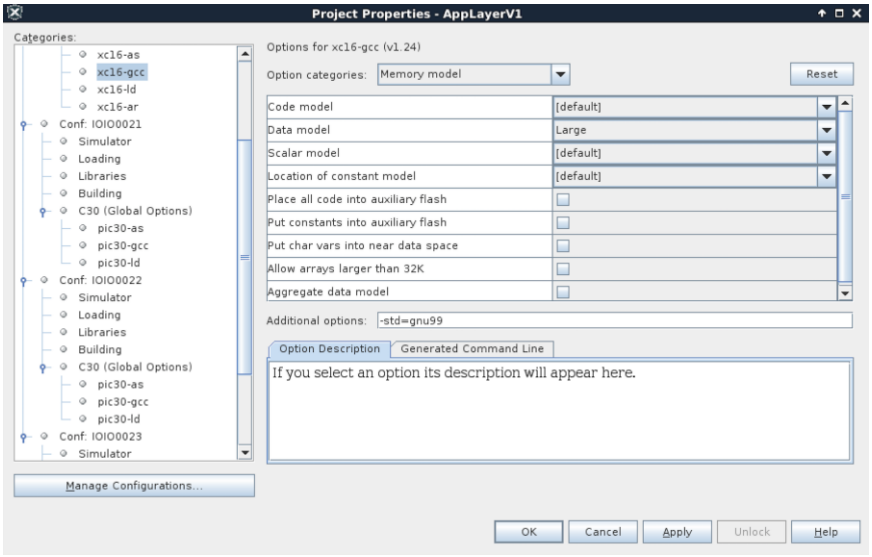
Tras cargar cada uno de los proyectos, habrá que modificar las propiedades de cada uno de ellos, indicando en cada lugar el microcontrolador para el que se compilará el proyecto, para IOIO OTG el PIC24FJ26GB206. Esto se hará seleccionando el proyecto en Properties>Manage Configurations…. Para libusb será necesario seleccionar (marcar con "Set Active") PIC24FJ26GB206_OTG. Para libconn, libbtstack y libadb se seleccionará PIC24FJ26GB206. Finalmente, en AppLayerV1 se establecerá como configuración IOIO0030:



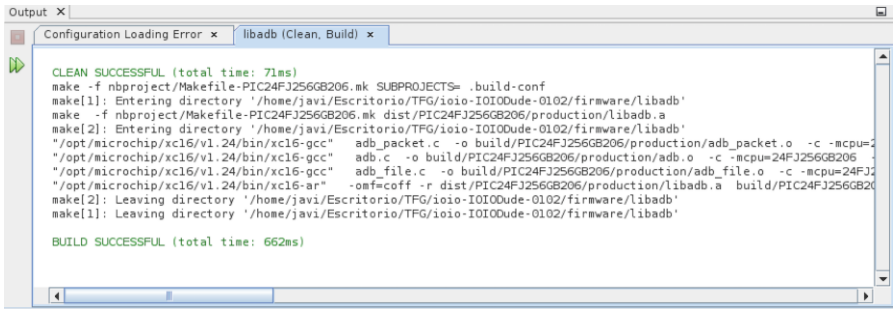
Por último, en las opciones del compilador de C (xc16-gcc) dentro de "Optimizations" habrá que establecer "Optimization level" a 1 en cada uno de los proyectos:



Antes de finalizar con la configuración del proyecto, hay que tener en cuenta que si se utilizan arrays de una longitud considerable como en este proyecto, será necesario modificar el modelo de datos (*Data model*) del compilador para poder repartir todos esos datos en la memoria del microcontrolador. En la siguiente imagen se muestra cómo se ha configurado para este proyecto:



A partir de este momento, los proyectos ya estarían configurados y sólo sería necesario construir cada uno de ellos, terminando con AppLayerV1. Esto se consigue seleccionando en cada proyecto la opción "Clean and Build". De esta forma, se podrá observar una salida como la siguiente:



Tras construir el proyecto principal, se obtiene un fichero tipo .hex en el directorio dist (MPLABXProjects/firmware/app_layer_v1/dist/IOIO0030/production/) que contiene el ejecutable del firmware que posteriormente se ejecutará en IOIO. Sin embargo, es necesario modificar el formato de este fichero convirtiéndolo a un fichero tipo .ioio, mucho más compacto que el .hex y más fácil de procesar por el bootloader de IOIO.

Para tal fin, se proporciona las herramientas hex2ioio y make-ioio-bundle, dentro del directorio "tools" en https://github.com/ytai/ioio. La herramienta a ejecutar para ello será la segunda, que hace uso, asimismo, de la primera. Así pues, en primer lugar h abrá que compilar el ejecutable de hex2ioio, disponible en tools/hex2ioio, ya que solo se proporciona el código fuente de la herramienta y un Makefile para su compilación. De esta forma, simplemente ejecutando la orden make desde el terminal se conseguirá el ejecutable necesario para poder obtener el fichero .ioio:

```
javi@debian:~/TFG/ioio-IOIODude-0102/tools/hex2ioio$ ls -l
total 48
-rwxr-xr-x 1 javi javi 33560 may 25 18:21 hex2ioio
-rw-r--r-- 1 javi javi  5359 sep 15  2014 hex2ioio.cc
-rw-r--r-- 1 javi javi    14 sep 15  2014 Makefile
```

El siguiente paso sería, por tanto, ejecutar la herramienta make-ioio-bundle. Como argumentos habrá que pasarle el dirctorio dist donde se encuentra el fichero .hex, un nombre para el fichero resultante y la configuración usada, que como se dijo anteriormente sería IOIO0030. Como resultado se obtendría un fichero .zip al que habría que renombrar como .ioio. En la siguiente figura se muestran cada uno de los comandos ejecutados:

```
javi@debian:~/TFG/ioio-IOIODude-0102$ ./tools/make-ioio-bundle
Usage: make-ioio-bundle dist-dir out config1 config2 ...
javi@debian:~/TFG/ioio-IOIODude-0102$ ./tools/make-ioio-bundle firmware/app_laye
r_v1/dist/ IOIOFW IOIO0030 IOIO0030
IOIO0030
Read 470 blocks
Success!
IOIO0030
Read 470 blocks
Success!
updating: IOIO0030.ioio (deflated 41%)
javi@debian:~/TFG/ioio-IOIODude-0102$ mv IOIOFW.zip IOIOFW.ioio
javi@debian:~/TFG/ioio-IOIODude-0102$ ls -l | grep IOIO
-rw-r--r--  1 javi javi 54359 may 25 18:24 IOIOFW.ioio
```

Una vez que se tiene el firmware de IOIO en el formato adecuado hay que transferirlo al microcontrolador, para lo cual será necesario conectarlo al ordenador. Habrá que tener en cuenta lo siguiente:

- El switch "host" de IOIO debe estar en modo automático (posición A), de forma que IOIO pueda detectar que tiene que actuar como un dispositivo USB.
- Se debe conectar IOIO al PC utilizando un cable USB micro-B. Además, IOIO debe estar en modo *bootloader*, lo cual se obtiene conectando el pin BOOT a tierra (GND) antes de conectar IOIO al ordenador. Tras conectarlo, se encenderá el led amarillo de IOIO e inmediatamente habrá que desconectar el pin boot de tierra. El led parpadeará tres veces, indicando que ha entrado en modo bootloader.
- Para que el computador detecte IOIO conectado a un USB, en Linux hay que añadir una regla udev descargable desde https://github.com/ytai/ioio/raw/master/driver/50-ioio.rules. En la siguiente imagen se muestra el procedimiento seguido para añadir la regla y cómo ver que el SO ha detectado que IOIO está conectado:

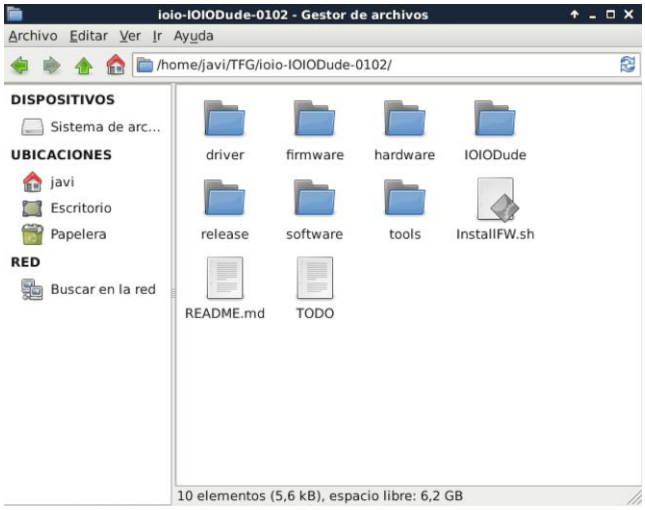
```
root@debian:~# mv /home/javi/50-ioio.rules /etc/udev/rules.d/
root@debian:~# udevadm control --reload-rules
root@debian:~# ls /dev/IOIO*
ls: no se puede acceder a /dev/IOIO*: No existe el fichero o el directorio
root@debian:~# ls /dev/IOIO*
/dev/IOIO0
```

Finalmente, para transferir el firmware al microcontrolador se utilizará la herramienta IOIODude accesible a través de https://github.com/ytai/ioio/raw/master/release/apps/IOIODude-0102.zip. Descomprimiendo el archivo y dando permisos de ejecución a ioiodude se podrá ejecutar la herramienta y ver las opciones disponibles. En la siguiente figura se muestran los pasos seguidos para instalar en IOIO el firmware compilado anteriormente:

```
javi@debian:~/TFG/ioio-IOIODude-0102$ ./IOIODude/ioiodude
IOIODude V1.2

Usage:
ioiodude <options> versions
ioiodude <options> fingerprint
ioiodude <options> write <ioioapp>

Valid options are:
--port=<name> The serial port where the IOIO is connected.
--reset Reset the IOIO out of bootloader mode when done.
--force Bypass fingerprint matching and force writing.
javi@debian:~/TFG/ioio-IOIODude-0102$ ./IOIODude/ioiodude --port=IOIO0 versions
IOIO Bootloader detected.

Hardware version: SPRK0020
Bootloader version: IOIO0400
Platform version: IOIO0030
javi@debian:~/TFG/ioio-IOIODude-0102$ ./IOIODude/ioiodude --port=IOIO0 --reset --force write release/firmwar
e/application/IOIOFW.ioio
Writing image...
[####################################]
Writing fingerprint...
Done.
```

Para hacer más liviano el proceso de carga del firmware en IOIO, se ha creado un script para Linux que

permita obtener el fichero con el firmware y cargarlo directamente en IOIO sin tener que ejecutar comando por comando. Para su funcionamiento solo es necesario que los directorios se encuentren organizados de la misma forma que en el momento de la descarga e incluir el directorio IOIODude descargado anteriormente. De tal forma se tendrá lo siguiente:



En cualquier caso, se ha adjuntado el código del script, por lo que se podría cambiar esta organización.

Al ejecutar el script se muestra un menú con 4 opciones: instalar un firmware simple desde Blink, instalar el firmware desde App Layer, instalar la última versión disponible del firmware o salir. En la siguiente captura se muestra cómo sería la ejecución para instalar un nuevo firmware desde App Layer:

```
Installation menu for IOIO firmware:
1) Install new firmware from Blink
2) Install new firmware from App Layer
3) Install last official firmware last version
4) Exit
Option choosen:
2
Please, introduce the name of the firmware file:
IOIOFW
IOIO0030
Read 470 blocks
Success!
  adding: IOIO0030.ioio (deflated 41%)
Please, connect IOIO to the computer and press Enter

IOIO Bootloader detected.

Hardware version: SPRK0020
Bootloader version: IOIO0400
Platform version: IOIO0030
Please, press enter to install the firmware

Writing image...
[########################################]
Writing fingerprint...
Done.
```

De esta forma, ya se tendría un firmware con modificaciones propias cargado en IOIO.

2. JDK

El siguiente paso sería realizar los mismos cambios en la librería de Android, para lo que será necesario instalar Eclipse. Antes de proceder con ello, habrá que instalar la versión de JDK (*Java Development Kit*) necesaria, en este caso la versión 1.7.0_80, disponible a través de http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html.

El archivo descargado habrá que descomprimirlo en el directorio /opt (opcional). Una vez descomprimido, hay que actualizar las variables de entorno. Para lo cual, será necesario añadir las siguientes líneas a los ficheros /etc/profile y /home/nombre_usuario/.bashrc:

```
JAVA_HOME=/opt/jre1.7.0_80
export JAVA_HOME

PATH=$JAVA_HOME/bin:$PATH
export PATH
```

Ahora habrá que indicar al SO dónde se encuentra el JDK/JRE mediante la herramienta update-alternatives:

```
root@debian:/home/javi/Escritorio/TFG# update-alternatives --install "/usr/bin/j
ava" "java" /opt/jdk1.7.0_80/bin/java
java            javadoc         javah           java-rmi.cgi
javac           javafxpackager  javap           javaws
root@debian:/home/javi/Escritorio/TFG# update-alternatives --install "/usr/bin/j
ava" "java" "/opt/jdk1.7.0_80/bin/java" 1
update-alternatives: utilizando /opt/jdk1.7.0_80/bin/java para proveer /usr/bin/
java (java) en modo automático
root@debian:/home/javi/Escritorio/TFG# update-alternatives --install "/usr/bin/j
avac" "javac" "/opt/jdk1.7.0_80/bin/javac" 1
update-alternatives: utilizando /opt/jdk1.7.0_80/bin/javac para proveer /usr/bin
/javac (javac) en modo automático
root@debian:/home/javi/Escritorio/TFG# update-alternatives --install "/usr/bin/j
avaws" "javaws" "/opt/jdk1.7.0_80/bin/javaws" 1
update-alternatives: utilizando /opt/jdk1.7.0_80/bin/javaws para proveer /usr/bi
n/javaws (javaws) en modo automático
root@debian:/home/javi/Escritorio/TFG# ls -la /usr/bin | grep java
lrwxrwxrwx  1 root   root         22 abr 27 18:18 java -> /etc/alternatives/java
lrwxrwxrwx  1 root   root         23 abr 27 18:18 javac -> /etc/alternatives/jav
ac
lrwxrwxrwx  1 root   root         24 abr 27 18:19 javaws -> /etc/alternatives/ja
vaws
```

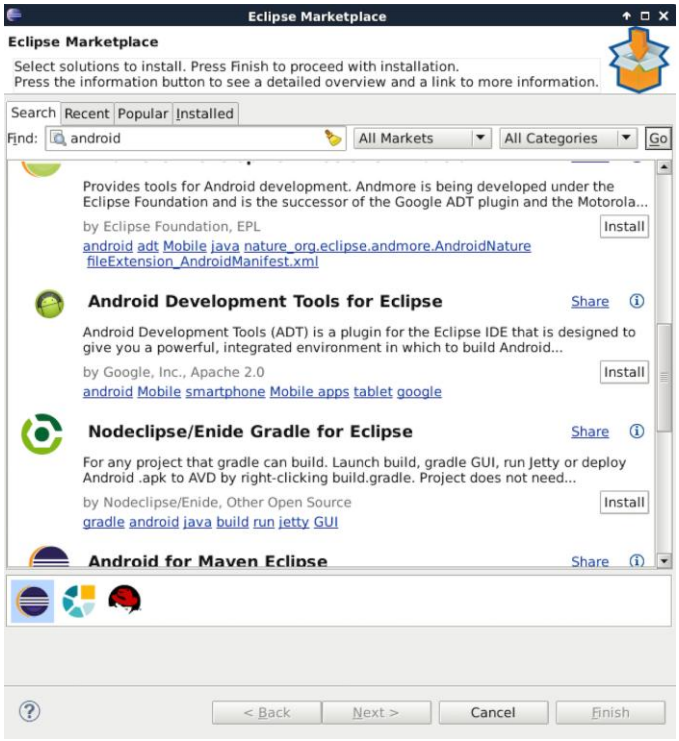
Finalmente, la instalación concluiría con el reinicio del equipo. De la siguiente forma se puede comprobar que se ha instalado correctamente:

```
root@debian:~# java -version
java version "1.7.0_80"
Java(TM) SE Runtime Environment (build 1.7.0_80-b15)
Java HotSpot(TM) 64-Bit Server VM (build 24.80-b11, mixed mode)
root@debian:~# javac -version
javac 1.7.0_80
```

3. Eclipse

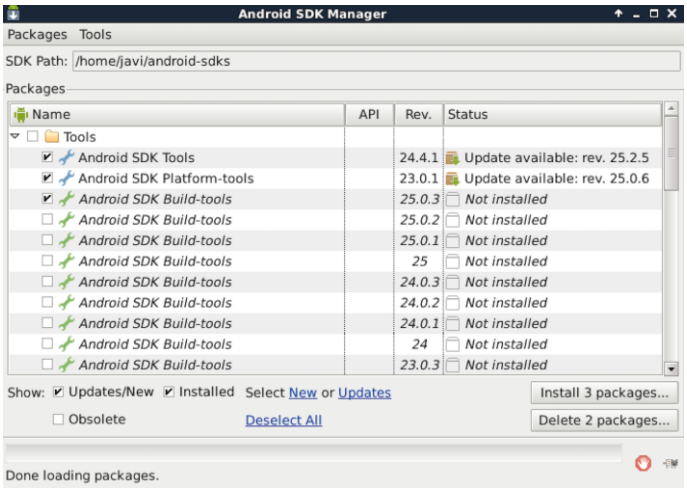
La versión de Eclipse a descargar será Eclipse Juno for Mobile Developers, accesible desde http://www.eclipse.org/downloads/packages/release/Juno/SR2. Una vez descargado, en primer lugar habrá que descomprimirlo en el directorio /opt y lanzar el ejecutable de Eclipse que se encuentra dentro del directorio resultante.

Una vez dentro de Eclipse, se descargarán las herramientas de desarrollo de Android (*Android Development Tools, ADT*). Para ello, habrá que ir a Help>Eclipse Marketplace y buscar las ADT, en concreto habrá que isntalar *Android Development Tools for Eclipse*:

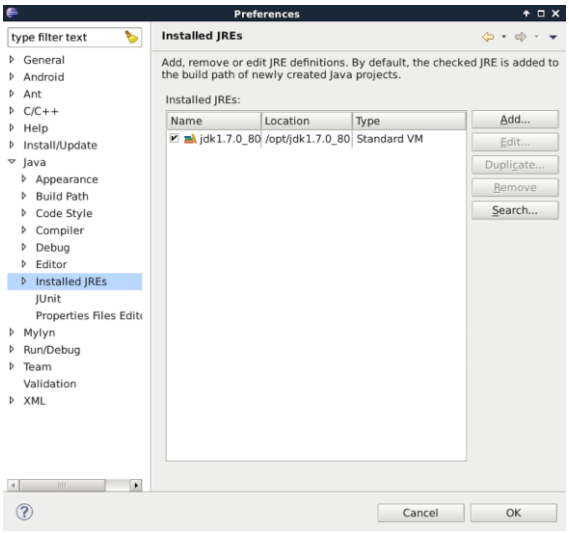


El siguiente paso será instalar el SDK de Android. Únicamente se instalarán las herramientas para Android 2.2 y después se añadirán algunas más necesarias para el proyecto:

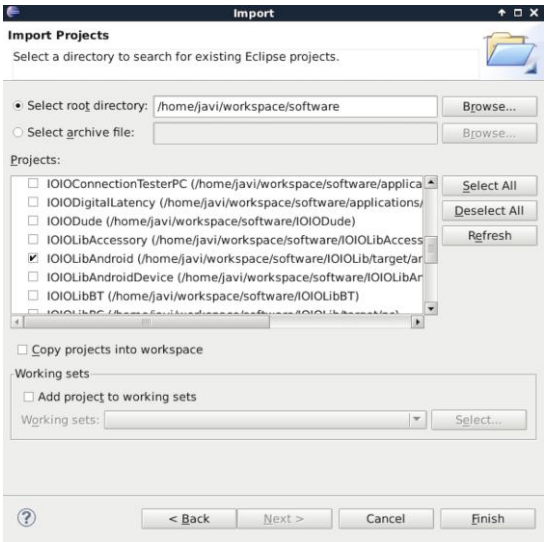
Cuando se instale el SDK, habrá que ir al SDK Manager e instalar las Android SDK Tools, Platform-tools y Build-tools, además de las API 7, 16, 19 y 21(sólo Platform SDK y Google API), necesarias para la librería de IOIO.



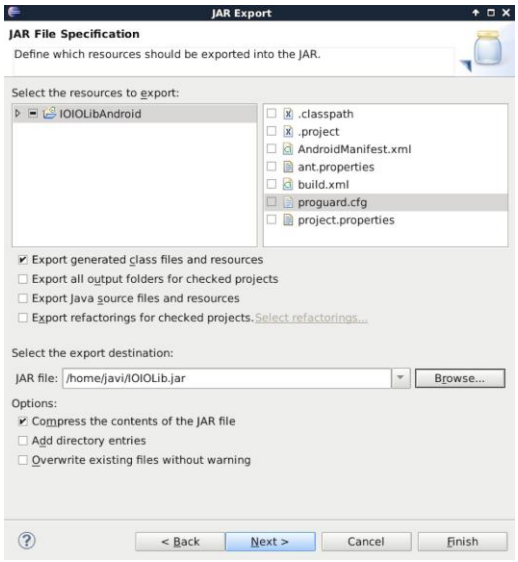
Seguidamente, es necesario comprobar que el JDK que está usando es el que se instaló en el paso anterior. Para ello, dentro de Window>Preferences, en Java>Installed JREs, se puede seleccionar conociendo la ubicación de la instalación.



Una vez instalado y configurado, hay que importar el proyecto. Como las únicas funciones que se van a modificar serán las referentes a Android, y la transmisión será por cable, solo será necesario importar IOIOLibAndroid. Así, hay que seleccionar File>Import>General>Existing Projects into Workspace y seleccionarlo dentro de la carpeta software.



El siguiente paso será obtener un archivo JAR (*Java Archive*) que se utilizará posteriormente en la programación de la aplicación Android y que incluirá todas las funciones de IOIO. Seleccionando Export…>Java>JAR file aparecerá la siguiente ventana y se obtendrá el fichero donde se especifique:



4. Android Studio

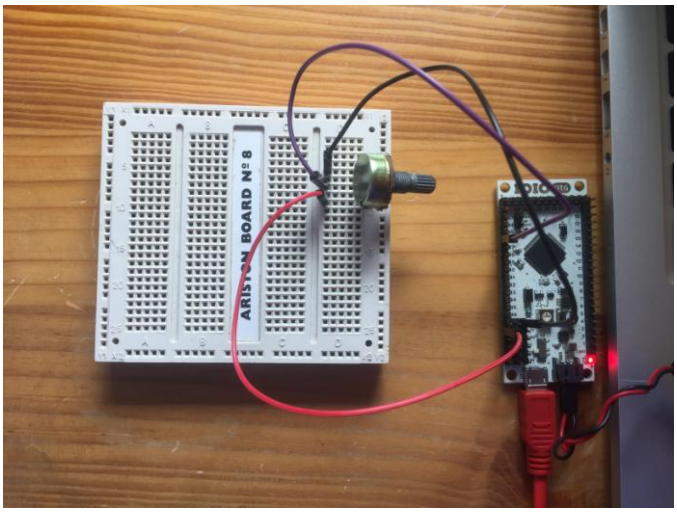
La versión de Android Studio que se ha utilizado para el desarrollo de la aplicación es la 1.1, y se puede obtener a través del siguiente enlace http://tools.android.com/download/studio/canary/1-1-0. Al igual que antes, el archivo descargado se descomprimirá en el directorio /opt. Para la instalación, se elegirá la opción *Custom*, y se proporcionará la ruta del SDK instalado anteriormente con Eclipse.

Para comenzar a desarrollar aplicaciones para IOIO en Android Studio simplemente habrá que incorporar la librería que se obtuvo anteriormente en Eclipse, dar permisos de Internet y Bluetooth en el manifest y emplear la estructura de clases y métodos de IOIO que se vio en capítulos anteriores.

ANEXO D: MANUAL DE USUARIO

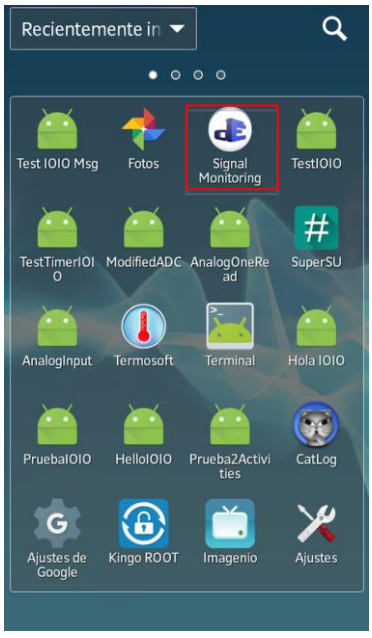
En este anexo se proporcionará una guía al usuario final del sistema de monitorización de señales, de forma que se indicará paso por paso cómo utilizar IOIO y la aplicación creada para monitorizar una señal.

Para comenzar, el usuario deberá conectar el sensor analógico a IOIO, concretamente al pin 36, tal y como se puede ver en la siguiente imagen:

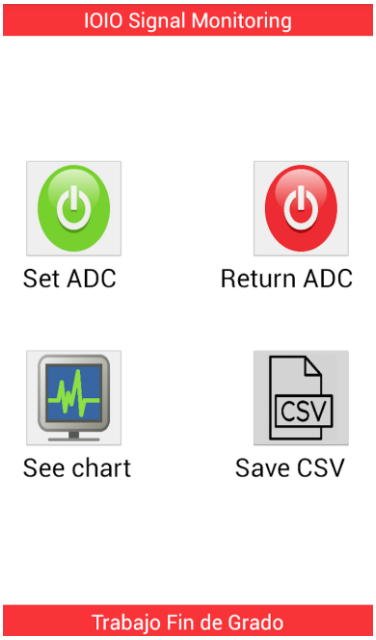


Asimismo, se conectará la alimentación mediante la salida USB de cualquier ordenador, o batería portátil, siempre que no exceda de 1,5 A y 10 V.

Una vez hecho esto, se conecta el USB micro-A a IOIO y el USB micro-B al terminal Android, donde se deberá lanzar la aplicación *Signal Monitoring*.

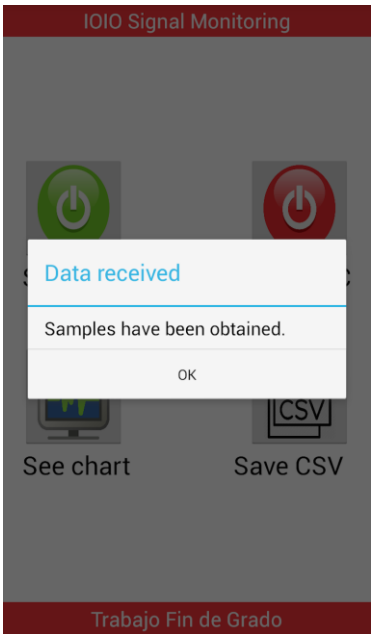


Una vez dentro, la aplicación se presenta de la siguiente forma:



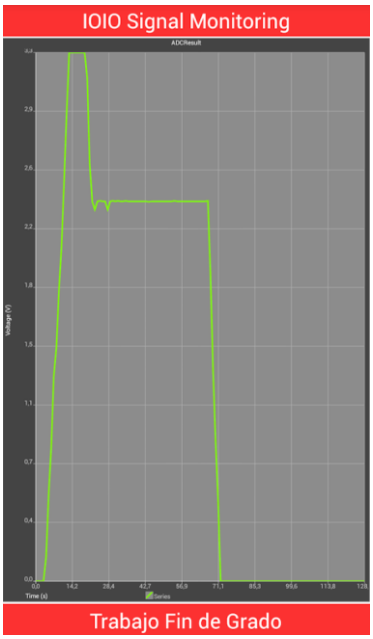
Para comenzar el muestreo de la señal, se pulsará sobre el botón *Set ADC*. Una vez que comience, el terminal se puede retirar y IOIO estará muestreando hasta 20 minutos o hasta que el usuario decida pararlo.

Para parar, el terminal deberá estar conectado a IOIO y se debe pulsar el botón Return ADC. Si todo es correcto, cuando se reciban las muestras, aparecerá el siguiente mensaje en pantalla:



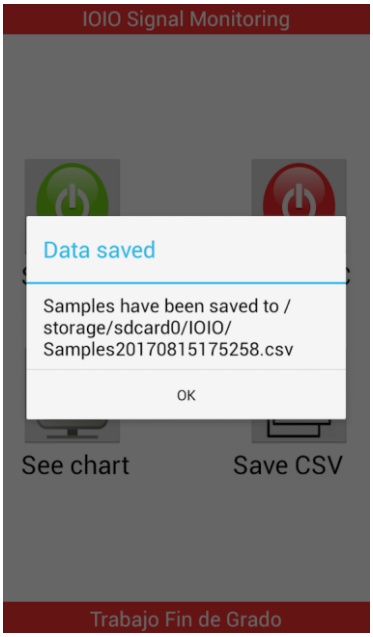
A partir de este momento, los resultados se pueden observar de dos formas diferentes:

1) Pulsando el botón *See chart*. Este botón redirigirá a otra actividad que mostrará la variación de la señal de entrada respecto al tiempo.



2) Pulsando el botón *Save CSV*. De esta forma, se almacenará un fichero en formato CSV que almacenará las muestras sin formato, es decir, mostrará lo que había en el registro del convertidor en el momento de la conversión de la muestra. Así, posteriormente el usuario podrá decidir cómo estudiar o qué hacer con los datos sin perder información por aproximaciones.

   Al pulsar el botón, se mostrará un mensaje indicando dónde se ha almacenado el fichero y su nombre:



Conectando el terminal Android a un ordenador para examinar su memoria interna, se puede ver el fichero guardado. Este se almacena por defecto en una carpeta denominada IOIO, y toma su nombre de la forma añadiendo a *Samples* la fecha y la hora. El fichero obtenido tendrá la siguiente forma: