View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Documental de la Universidad de Valladolid

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería Electrónica Industrial y Automática

DISEÑO DE UN PROGRAMA EN ANDROID PARA EL CONTROL DE ARDUINO

Autor:

Frades Estévez, Jesús Alberto

Tutor:

Plaza Pérez, Francisco
Departamento de Tecnología Electrónica

Valladolid, Septiembre 2015.

Resumen

Desarrollo de una aplicación en Android para el control de la tarjeta Arduino Yún.

La comunicación entre los dos dispositivos se realizará a través de una red WiFi (IEEE 802.11), permitiendo el control de las entradas/salidas digitales así como las entradas analógicas del modulo Arduino Yún. Posibilitando el control de dichas señales de forma remota.

También se realizará la aplicación necesaria en Arduino, para el correcto funcionamiento del sistema.

Palabras clave: Android, Arduino, programación, comunicación, WiFi.

Índice general

Resumen 3
Índice general 5
Índice de ilustraciones 9
Índice de tablas 11
Glosario de términos y abreviaturas 13
Introducción 15
Objetivos 16
Estudio previo 17
Aplicaciones 20

Capítulo 1 Descripción del software 21
1.1 Android 21
1.1.1 Entorno de desarrollo de Android 22
1.1.2 Base inicial de la aplicación 23
1.1.3 Componentes 24
1.1.4 Ficheros y carpetas de un proyecto Android 27
1.1.5 Creación de un proyecto Android en Eclipse 29
1.2 Arduino 32
1.2.1 Entorno de desarrollo de Arduino 32
1.2.2 Configuración del IDE 34
1.2.3 Estructura de un programa 35

Capítulo 2 Descripción del hardware 37
2.1 Arduino Yun 37
2.2 Configuración inicial del Arduino Yun 40
2.3 Datos analógicos y digitales 43

Capítulo 3 Programación 45
3.1 Diseño de la interfaz de usuario: *Vistas* y *Layouts* 45
3.2 Navigation Drawer o Menú de navegación 55
3.2.1 Adaptador para el ListView 55
3.2.2 Eventos y opciones del Navigation Drawer 57
3.3 Fragmentos 60
3.3.1 Ciclo de vida de un Fragmento 61
3.3.2 Implementación de un Fragmento 63
3.3.3 Gestión de los Fragmentos 64
3.3.4 Creación de un Fragmento 64
3.3.5 Comunicación con Fragmentos 68
3.4 Hilos de ejecución. Tareas asíncronas 71
3.5 Protocolo HTTP 72
3.6 Almacenamiento de datos 74
3.7 AndroidManifest.xml 77
3.8 Código Arduino. Librería Bridge 79
3.9 Funciones de Arduino 83

Capítulo 4 Funcionamiento 85
4.1 Instalación 85
4.2 Acceso a la aplicación 87

Capítulo 5 Caso práctico 93
5.1 Esquema general 93
5.2 Componentes electrónicos 93
5.2.1 Leds 94
5.2.2 Servomotores 97

5.2.3 Fotorresistencias 97

Capítulo 6 Conclusiones y trabajos futuros 99

6.1 Objetivos cumplidos.. 99

6.2 Mejoras y trabajos futuros 100

Bibliografía 101

Anexo A Guía de instalación de Eclipse 103

Anexo B Guía de instalación de Arduino 109

Anexo C Contenido del CD-ROM 111

Índice de ilustraciones

Ilustración 1: Arquitectura de Android 18
Ilustración 2: Placas de Arduino UNO, MEGA y YÚN 19
Ilustración 3: Comparativa entre menú lateral y pestañas 24
Ilustración 4: Estructura de un proyecto 27
Ilustración 5: Configuración inicial Android Application Project. 29
Ilustración 6: Definición de valores de un proyecto en Android. 30
Ilustración 7: Creación y diseño del icono de la aplicación 30
Ilustración 8: Creación de una actividad 31
Ilustración 9: Nombres de la activity, layout, fragment y NavigationDrawer. . 31
Ilustración 10: Entorno de desarrollo Android 32
Ilustración 11: Botonera del IDE Arduino 1.6.1 33
Ilustración 12: Monitor serie 34
Ilustración 13: Configuración de la placa y puerto en el IDE de Arduino 34
Ilustración 14: Bloque setup() de un sketch 35
Ilustración 15: Bloque loop() de un sketch 35
Ilustración 16: Comunicación entre Arduino y Linino (Linux) 37
Ilustración 17: Placa Arduino Yun 40
Ilustración 18: Red inalámbrica generada por Arduino Yun 41
Ilustración 19: Configuración inicial Arduino Yun 42
Ilustración 20: Parámetros de configuración de Arduino Yun 42
Ilustración 21: Interfaz de diseño. Graphical Layout 46
Ilustración 22: Interfaz de diseño. XML 46
Ilustración 23: Vista del Layout "Ajustes" 50
Ilustración 24: Vista layout controles 52
Ilustración 25: Layouts header.xml, tutorial.xml y splash.xml 53
Ilustración 26: Iconos del menú de navegación 58
Ilustración 27: Botones de acceso al menú 58
Ilustración 28: Aplicación en smartphone y tablet con *fragments*. 60

Ilustración 29: Paso de parámetros a través de la actividad de acogida. 68
Ilustración 30: Archivo .apk 85
Ilustración 31: Seguridad en la instalación de aplicaciones 86
Ilustración 32: Permisos de la aplicación 86
Ilustración 33: Icono de acceso a la aplicación 87
Ilustración 34: Imagen de lanzamiento de la aplicación 87
Ilustración 35: Ajustes y menú lateral de la aplicación 88
Ilustración 36: Vista “Controles” de la aplicación 89
Ilustración 37: Vista "Sensores" de la aplicación 89
Ilustración 39: Pantallas de ayuda de la aplicación 90
Ilustración 38: Iconos en la *ActionBar* 90
Ilustración 40: Almacenamiento externo con la configuración de los pines... 91
Ilustración 41: Esquema del cableado del caso práctico 94
Ilustración 42: Leds 95
Ilustración 43: Dimensiones de un led 95
Ilustración 44: Servomotor 97
Ilustración 45: LDR 98
Ilustración 46: Divisor de tensión 98
Ilustración 47: Paquete de descarga del entorno de desarrollo 103
Ilustración 49: Instalación y comprobación de Java 104
Ilustración 48: Espacio de trabajo de Eclipse 104
Ilustración 50: Descarga del paquete JDK de Java 105
Ilustración 51: Directorio del espacio de trabajo de Eclipse 105
Ilustración 52: Enlace de descarga SDK 106
Ilustración 53: Recursos SDK Android 106
Ilustración 54: Instalación del software 106
Ilustración 55: Localización del SDK de Android 107
Ilustración 56: Descarga del software 109
Ilustración 57: Administrador de dispositivos 110

Índice de tablas

Tabla 1: Ventajas y desventajas entre menú lateral y pestañas..................... 24
Tabla 2: Características del entorno Arduino 38
Tabla 3: Características del microprocesador Linux 38
Tabla 4: Caídas de tensión de los LEDs en función de color 95

Glosario de términos y abreviaturas

ADT	*Android Development Tools*
API	*Application Programming Interface*
ASCII	*Grupo de 255 caracteres definidos por el American Standard Code for Information Interchange*
AVD	*Android Virtual Device*
Baud	*Baudio. Unidad de medida de transmisión referida al numero de bits por segundo que transmite*
EEPROM	*Tipo de EPROM que puede ser borrada con una señal eléctrica*
EPROM	*Erasable Programmable Read-Only Memory*
GPS	*Graphic Processing System.*
HTML	*HyperText Markup Language*
IP	*Internet Protocol*
Kernel	*Núcleo de un SO. Maneja la memoria, archivos, periféricos, recursos..*
LDR	*Light Dependent Resistor*
LED	*Light Emitting Diode*
Log	*Registro de actividades que ocurren en un sistema o programa.*
Eclipse	*Software de programación para el desarrollo de aplicaciones.*
HTTP	*HyperText Transfer Protocol*

IEEE	*Instituto de ingenieros eléctricos y electrónicos.*
IDE	*Integrated Development Environment.*
IU	*Interfaz de usuario*
JAR	*Java ARchive*
JDK	*Java Development Kit*
JRE	*Java Runtime Environment*
JVM	*Java Virtual Machine*
PNG	*Portable Network Graphics*
REST	*Representational State Transfer*
PWM	*Pulse With Modulation*
SD	*Secure Digital*
SDK	*Software Development Kit*
SO	*Sistema operativo*
TCP	*Transmission Control Protocol*
UI	*User Interface*
URL	*Universal Resource Locator*
USB	*Universal Serial Bus*
WWW	*World Wide Web*
Wi-Fi Alliance	*Asociación compuesta por diversas empresas tecnológicas que ofrecen el estándar 802.11*
WLAN	*Wide Local Área Network*
XML	*Extensible Markup Language*

Introducción

La tecnología ha sido parte fundamental del desarrollo. Su continuo crecimiento, permite crear mejores herramientas útiles para facilitar y simplificar el tiempo y esfuerzo del trabajo.

Una de las tecnologías en auge es la informática. Su objetivo principal consiste en automatizar mediante equipos generalmente electrónicos, todo tipo de información. Para poder automatizar dicha información, la informática se basa en la realización de 3 tareas básicas:

- La entrada de información.
- El tratamiento de la información
- Salida de la información.

El sistema informático ha de estar dotado de algún medio por el cual se aporte la información necesaria, que más tarde el sistema interpretará y guardará y que posteriormente solicitaremos mediante algún medio de salida. La ciencia de la informática se desglosa en diversas ramas de la ciencia como la programación, la arquitectura de redes y computadores, la electrónica, la electricidad, la inteligencia artificial, etc., aspectos básicos que permiten desarrollar el software y hardware necesario en la informática.

De las ramas anteriores, en este proyecto nos centraremos en las tres primeras. La programación, a través de la cual podremos configurar y diseñar nuestro proyecto. En segundo lugar, la arquitectura de redes, donde conseguiremos la conexión e intercambio de datos entre los diferentes dispositivos electrónicos (Hardware). Y por último la electrónica, donde aplicaremos los conocimientos necesarios para mostrar ejemplos prácticos y útiles de este proyecto, estableciendo las oportunas conexiones.

Internet se le conoce como una gran red basada en la interconexión entre dispositivos creando una gran red de intercomunicaciones. Es un espacio donde se puede compartir cualquier producto, servicio o información con cualquier persona y lugar del mundo.

Para acceder al intercambio de datos a través de los protocolos TCP/IP que utiliza Internet, utilizamos la tecnología inalámbrica conocida también como *Wireless*. Esta elección se debe a su mayor comodidad a la hora de su instalación, el ahorro en componentes, la flexibilidad de no estar atados a un cable de forma rígida. La red utilizada para la comunicación es la red inalámbrica privada WLAN 802.11.

En 1997 fue lanzado el estándar 802.11 por parte del IEEE. Posteriormente en el año 1999 varias empresas crearon la asociación sin ánimo de lucro WECA con la finalidad de fomentar el desarrollo de dispositivos electrónicos compatibles con el estándar 802.11. En el año 2003 se rebautizo con el nombre Wi-Fi Alliance, nombre que utilizaremos en adelante para referirnos a esta tecnología.

WiFi, como se ha dicho anteriormente es una tecnología inalámbrica utilizada para conectar e intercambiar información entre dispositivos electrónicos sin necesidad de conectarlos mediante el uso de cables físicos.

Objetivos

El objetivo principal del proyecto es el diseño de una aplicación para el sistema operativo Android que permita controlar mediante WiFi el microcontrolador Arduino.

Para ello dispondremos de un Smartphone con sistema operativo Android donde instalaremos la aplicación y un microcontrolador como Arduino Yun con el que estableceremos la conexión y envío de datos. Los objetivos principales son:

- Conocer las principales características de los lenguajes Java y C.
- Conocer las principales características de Android y Arduino.
- Estudiar el entorno de desarrollo de Android y Arduino.
- Desarrollar la aplicación para Android.
- Programar el código para Arduino.
- Establecer la conexión y comunicación entre las dos plataformas.
- Envío y respuesta de datos para el control remoto.
- Lectura de sensores de la placa Arduino.
- Aplicación práctica.

Estudio previo

Como hemos explicado anteriormente, los dos pilares fundamentales de este proyecto son las dos plataformas, Android y Arduino.

La telefonía móvil está cambiando la sociedad actual de una forma tan significativa como lo ha hecho internet. Esta revolución no ha hecho más que empezar, los nuevos terminales ofrecen unas capacidades similares a un ordenador personal.

El lanzamiento de Android como nueva plataforma para el desarrollo de aplicaciones móviles ha causado una gran expectación. Android es una plataforma de desarrollo libre y de código abierto. El sistema operativo está basado en el núcleo Linux al que se le han hecho ciertas modificaciones para que pueda ejecutarse en teléfonos y terminales móviles. Su desarrollador, Google, ha escogido como lenguaje de programación Java, para asegurarse que las aplicaciones podrán ser ejecutadas en cualquier tipo de CPU, tanto presente como futuro. Esto se consigue gracias al concepto de máquina virtual, que explicaremos más adelante.

Android presenta una serie de características que lo hacen diferente, es el primer sistema operativo para móviles que combina en una misma solución las siguientes cualidades:

- Aceptables para cualquier tipo de hardware.
- Portabilidad asegurada.
- Arquitectura basada en componentes inspirados en internet.
- Filosofía de dispositivos siempre conectado a internet.
- Gran cantidad de servicios incorporados.
- Optimizado para baja potencia y poca memoria.

En cuanto a los orígenes, Google adquiere Android inc. en el año 2005. Ese mismo año empiezan a trabajar en la creación de una máquina virtual Java optimizada para móviles (Dalvik VM). Esta máquina permite ejecutar aplicaciones programadas en Java. Se encuentra en la capa de ejecución y ha sido diseñada para optimizar la memoria y los recursos de hardware en el entorno de los teléfonos móviles.

En cuanto a la arquitectura del sistema operativo Android, detallamos a través de la ilustración 1, los aspectos más importantes:

- **El núcleo Linux (*Kernel Linux*):**
  El núcleo de Android está formado por el sistema operativo Linux. Esta capa proporciona servicios como la seguridad, el manejo de la

memoria, el multiproceso, la pila de protocolos y el soporte de drivers para dispositivos. Esta capa es la única dependiente del hardware.

- Runtime de Android (*Android Runtime*):
  Está basado en el concepto de maquina virtual utilizado en Java. Dadas las limitaciones de los dispositivos en cuanto a poca memoria y procesador limitado, se creó la maquina virtual Dalvik, explicada anteriormente, para que respondiera mejor a esas limitaciones. A partir de Android L se utiliza la maquina virtual ART. Las características de la primera máquina virtual destacan la optimización de recursos a la hora de la ejecución de ficheros Dalvik ejecutables (*.dex*).

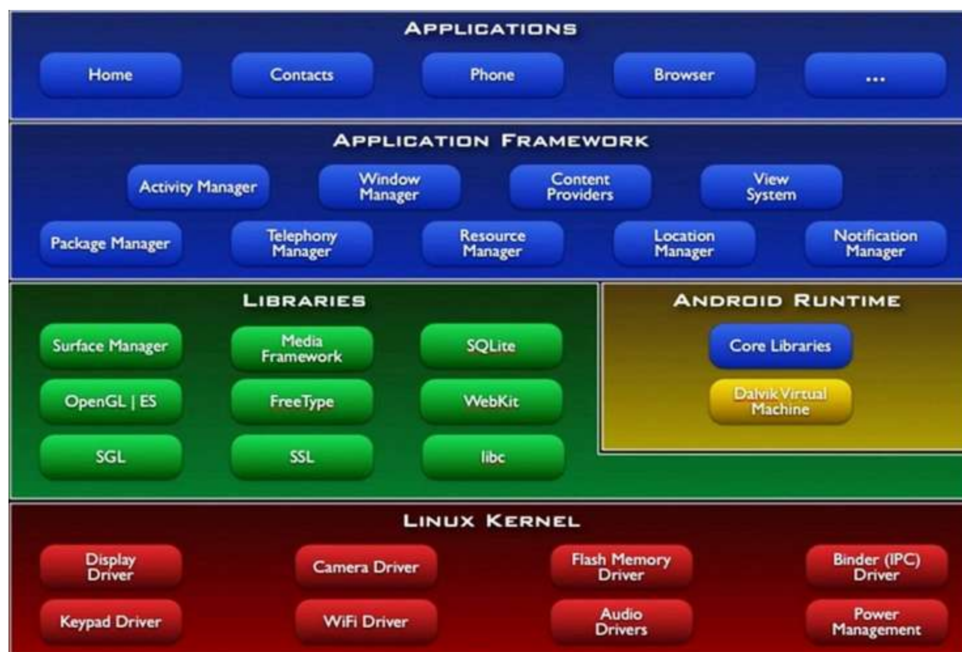


Ilustración 1: Arquitectura de Android

- Librerías nativas (*Libraries*):
  Siguiendo la ilustración 1, la siguiente capa después del kernel son las bibliotecas nativas de Android que incluyen un conjunto de librerías en C/C++ usadas en varios componentes de Android. Están compiladas en código nativo del procesador.

- Entorno de aplicación (*Application Framework*):
  Proporciona una plataforma de desarrollo libre para aplicaciones con gran riqueza e innovaciones (sensores, localización, servicios, barra de notificaciones, etc.). Esta capa ha sido diseñada para simplificar la reutilización de componentes. Los servicios más importantes que incluyen son: Views, Resource Manager, Activity Manager ,Notification Manager, Content Providers.

- Aplicaciones (*Applications*):
  Este nivel está formado por el conjunto de aplicaciones instaladas en una máquina Android. Todas las aplicaciones han de correr en la máquina virtual Dalvik para garantizar la seguridad del sistema. Normalmente las aplicaciones Android están escritas en Java.

En cuanto al otro pilar fundamental de este proyecto, Arduino, es una plataforma de hardware libre, basada en una placa con un microcontrolador y un entorno de desarrollo. Arduino dispone de una serie de tarjetas programables compuestas básicamente por un microcontrolador, un cristal oscilador y un regulador lineal de 5 voltios y que ofrece un cierto número de entradas y salidas tanto analógicas como digitales. La función del oscilador es proveer al microcontrolador de una serie de pulsos que permiten que pueda funcionar a una determinada velocidad. En general, las placas Arduino utilizan microcontroladores de 8 bits.

De una manera informal, podemos definir Arduino como un pequeño ordenador al que se le puede programar para que interactúe con el mundo real, bien ofreciendo salidas o reaccionando a una serie de entradas. Para poder programarlo, Arduino utiliza un lenguaje propio llamado *Arduino programming language* que se basa en el lenguaje de programación *Wiring.* Éste es un marco de programación de código abierto para microcontroladores.

Este microcontrolador puede tomar información del entorno a través de sus entradas analógicas y digitales utilizándolas de modo que reciban impulsos eléctricos. Mediante diferentes elementos se capta la realidad para transformar la corriente eléctrica como pueden ser acelerómetros, detectores de ultrasonidos, botones, etc.

En cuanto a los inicios de este dispositivo, Arduino se inició en el año 2005 como un proyecto para estudiantes. Posteriormente Google colaboró en el desarrollo del Kit Android ADK (Accesory Development Kit), una placa Arduino capaz de comunicarse directamente con teléfonos móviles inteligentes bajo el sistema operativo Android para que el teléfono controle entre otras muchas cosas luces, motores y sensores conectados a Arduino.

Ilustración 2: Placas de Arduino UNO, MEGA y YÚN

Aplicaciones

Uno de los términos más escuchados en torno a estas dos aplicaciones es el llamado "*Internet of things*" o "*Internet de las cosas*". Esta expresión define a los objetos que independientemente de su naturaleza o tamaño se encuentran conectados a Internet, permitiendo el almacenamiento, gestión y transmisión de información emitida por dichos objetos. Además nos permite automatizar diferentes procesos o actividades tanto laborales como ociosas. Es un concepto que se refiere a la interconexión digital de objetos cotidianos con Internet.

El desarrollo e implantación de Internet en la sociedad es considerado como la primera revolución digital. La segunda, es la introducción del *Internet de las cosas*. Las tecnologías de software y hardware en las que se apoya están totalmente desarrolladas. Tecnologías como Big data, Smart Cities, fibra óptica, comunicaciones Wireless entre las más conocidas, son ejemplos de tecnologías disponibles y que forman parte de esta segunda revolución.

Una de las principales ventajas que presenta, es la capacidad de poder ser aplicada en diferentes áreas y sectores como el sanitario, militar, empresas de servicio, sector agrario, ganadería, centros de educación, etc.

La entrada de Internet en el hogar también nos facilita los hábitos costumbres y rutinas que empleamos en realizar ciertas tareas. Tareas como el riego de las plantas, el encendido y apagado de luces o electrodomésticos, programado o controlado por el teléfono móvil y otras muchas comodidades son las que nos ofrece *Internet de las cosas.*

Conociendo los dispositivos de Android y Arduino, una parte del presente proyecto será dedicada a la demostración de un caso práctico aplicable a unas prácticas sencillas de laboratorio para aprender el manejo básico tanto de Android y sobre todo, de Arduino.

Capítulo 1
Descripción del software

En el primer capítulo describiremos las características del software que utilizaremos en el proyecto. Mostraremos y explicaremos los entornos de desarrollo, estructuras y lenguajes de programación necesarios para el correcto funcionamiento, tanto para Android como para Arduino.

1.1 Android

Como ya vimos en el estudio previo, Android es una plataforma de desarrollo libre y de código abierto y con una arquitectura compleja.

Dispone de una gran cantidad de servicios como lectores de códigos de barras, servicios de GPS entre otros muchos, incluidos en una base de datos que servirá para mantener la información de nuestras aplicaciones actualizadas. Con cada nueva versión del sistema operativo, Google pone a disposición de los programadores la información necesaria para el uso de las nuevas funcionalidades.

Este sistema operativo está pensado para dispositivos con poca capacidad de proceso, poca memoria y poca batería. En Android existen unos ciclos de vida para las aplicaciones y la gestión de este ciclo de vida es llevada a cabo desde el mismo sistema operativo. Esto hace que no nos tengamos que preocupar en cerrar algunas aplicaciones cuando queramos lanzar un nuevo programa. El sistema operativo ya se encarga por nosotros, liberando espacio o durmiendo las aplicaciones que no estén en uso.

En esta sección se mostrarán las herramientas utilizadas para el desarrollo de la aplicación.

1.1.1 Entorno de desarrollo de Android

Para comenzar a desarrollar una aplicación de forma nativa, debemos configurar un entorno de trabajo para implementar nuestras aplicaciones. Para ello, instalamos el paquete Eclipse ADT Bundle[1].

A través de la página web, *http://developer.android.com,* creada por Google, podemos descargarnos este paquete de forma gratuita. Incluye:

- Eclipse. Programa informático que proporciona una serie de herramientas para el desarrollo del software. Además de la construcción de las aplicaciones en Android, también permite realizar programas en Java, C++ o creaciones de sitios web.

- ADT Plugin. Complemento necesario para que podamos desarrollar las aplicaciones en Eclipse. Incluye bibliotecas propias para el desarrollo de Android.

- Android SDK Tools. Incluye más herramientas para compilar el código, depurar la aplicación, o probarla en dispositivos virtuales, AVD. Además, contiene unas herramientas diferentes para cada versión disponible en Android, denominadas API.

Esta interfaz de aplicación de Android es un conjunto de subrutinas, funciones y procedimientos que ofrece una cierta biblioteca para ser utilizado por un software. De una manera más común y familiar es como conocemos las versiones de Android. Cada nueva versión, ofrece nuevas funciones que pueden ser implementadas, así de esta forma, una actualización reciente, evita procesos o rutinas obsoletas y fallos en el sistema operativo.

Estas son las versiones que ha desarrollado Android desde los inicios:

- Android 1.0 Nivel de API 1 (Septiembre 2008)
- Android 1.1 Nivel de API 2 (Febrero 2009)
- Android 1.5 Nivel de API 3 (Abril 2009) "*Cupcake*"
- Android 1.6 Nivel de API 4 (Septiembre 2009) "*Donut*"
- Android 2.0 Nivel de API 5 (Octubre 2009) "*Éclair*"
- Android 2.1 Nivel de API 7 (Enero 2010) "*Éclair*"
- Android 2.2 Nivel de API 8 (Mayo 2010) "*Froyo*"
- Android 2.3 Nivel de API 9 (Diciembre 2010) "*Gingerbread*"
- Android 3.0 Nivel de API 11 (Febrero 2011) "*HoneyComb*"

[1] Actualmente existe un nuevo software de desarrollo, Android Studio. Este programa que estaba en versión beta cuando se inició el presente proyecto, fue sustituido por Eclipse, debido a la falta de herramientas y al no estar completa la documentación.

- Android 3.1 Nivel de API 12 (Mayo 2011) “*HoneyComb*”
- Android 3.2 Nivel de API 13 (Julio 2011) “*HoneyComb*”
- Android 4.0 Nivel de API 14 (Octubre 2011) “*Ice Cream Sandwich*”
- Android 4.0.3 Nivel de API 15 (Diciembre 2011) “*Ice Cream Sandwich*”
- Android 4.1 Nivel de API 16 (Julio 2012) “*Jelly Bean*”
- Android 4.2 Nivel de API 17 (Noviembre 2012) “*Jelly Bean*”
- Android 4.3 Nivel de API 18 (Julio 2013) “*Jelly Bean*”
- Android 4.4 Nivel de API 19 (Octubre 2013) “*KitKat*”
- Android 5.0 Nivel de API 21 (Noviembre 2014) “*Lollipop*”

Éste es el primer paso que debemos tomar a la hora de comenzar a programar la aplicación. Elegir una versión para que ésta pueda ofrecerse a un gran número de dispositivos en el mercado y que esté lo suficientemente actualizada para ofrecer un diseño novedoso y un funcionamiento correcto.

En el Anexo A, se recoge la documentación necesaria de cómo instalar paso a paso este entorno de desarrollo.

1.1.2 Base inicial de la aplicación

Antes de iniciarse en la programación es imprescindible tener una idea clara del diseño. En este apartado mostraremos las ideas iniciales necesarios para comenzar a programar nuestra aplicación.

En primer lugar, debemos adaptarla al objetivo de nuestro proyecto. Como características de la interactuación que nos ofrece Arduino, implementaremos tres interfaces distintas. Una de ellas para definir los ajustes que se aplicarán a la tarjeta de Arduino, otra vista para visualizar el control de los botones analógicos y digitales y una última para mostrar la lectura de sensores.

Con la estructura central pensada, aplicamos el diseño que tendrá nuestra aplicación. A través de la web de Google, encontramos documentación de todos los estilos que se pueden aplicar.

En el siguiente apartado se explicarán en detalle la estructura y componentes que contiene la aplicación. Hasta el momento, destacamos el uso de fragmentos frente actividades y el uso del menú lateral frente a las pestañas.

A continuación, la ilustración 3 muestra las diferencias mencionadas anteriormente, mientras que la tabla 1 nos enseña las ventajas y desventajas que nos ofrece cada uno de los componentes posibles a utilizar.

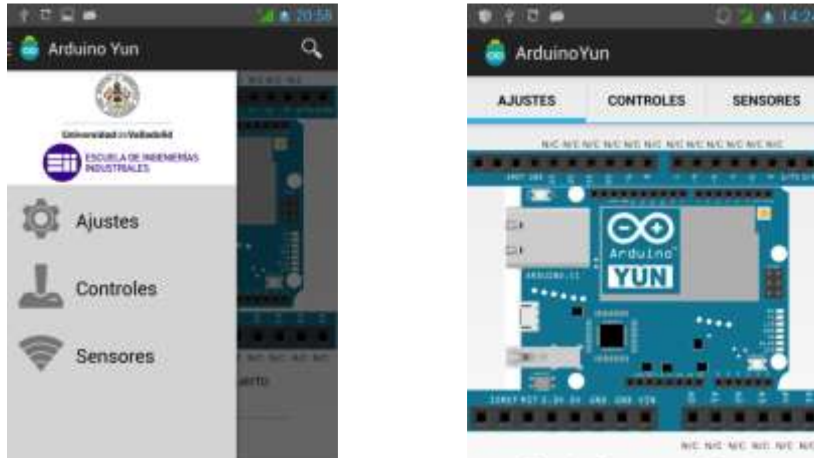


Ilustración 3: Comparativa entre menú lateral y pestañas

	Ventajas	Desventajas
Navigation Drawer o menu lateral	Diseño novedoso y visual. Mejor implementación dinámica entre las diferentes opciones. No ocupa espacio al estar siempre oculta.	Desplegar el menú cada vez que se quiera acceder a otra vista. No se puede implementar en versiones antiguas.
Tabs o Pestañas	Acceso rápido a otras actividades.	Ocupa espacio en la pantalla al estar siempre visible. En las versiones actuales está obsoleta.

Tabla 1: Ventajas y desventajas entre menú lateral y pestañas

1.1.3 Componentes

En este apartado veremos las piezas que Android pone a nuestra disposición, para que mediante su combinación, podamos crear aplicaciones. Para escribir una aplicación en Android podemos utilizar cinco bloques fundamentalmente denominados:

- **Activity** (Actividad)
- **Broadcast Intent Receiver** (Receptor de emisiones de intentos)
- **Sevice** (Servicio)
- **Content Provider** (Proveedor de contenido)
- **Fragment** (Fragmento)

Activity

Corresponde a una ventana o a un cuadro de dialogo en una aplicación de escritorio. Un *Activity* es una clase donde mostraremos *Views* (Vistas) para generar la interfaz de usuario y responderemos a eventos que se realicen sobre ella.

Son entidades independientes, capaces de llamarse entre ellas, pasándose parámetros y recibiendo respuestas, de modo que su funcionamiento sea un conjunto. Cada vez que una *Activity* llama a otra, la actividad que crea la llamada se introduce en una pila siguiendo el orden LIFO (*Last Input First Output,* último en entrar primero en salir), así el usuario pulsando el botón de "volver atrás" del teléfono, podrá recuperar la actividad anterior. Esto es una característica muy común de las actividades, a pesar de que el programador puede anular esa acción eliminando de la lista las actividades que menos interese.

A cada *Activity* se le asigna una ventana sobre la que dibujar una interfaz de usuario. En esta ventana el programador define qué elementos visuales y en qué lugar se van a mostrar. Este contenido se indica de un modo jerárquico mediante vistas, que son objetos que implementan la clase *View* (Vista). Esta clase es el nexo de unión entre las *Activity* y el usuario, ya que se encargan de recibir los eventos realizados y saber cómo va a responder cada uno de los elementos gráficos que derivan de la clase *View.*

Broadcast Intent Receivers

Un *Broadcast Intent Receivers* o receptor de mensajes, es un componente que simplemente se encarga de recibir y reaccionar frente a ciertos mensajes emitidos por el sistema. Para que este bloque sea accesible al sistema, es necesario registrarlo en el fichero *AndroidManifest.xml.* (En el siguiente apartado se explicará en detalle la función de este fichero).

Service

Las actividades tienen un periodo de vida corto y pueden estar ejecutándose y al poco tiempo ser desechadas. Los *Services* (servicios) están diseñados para mantenerse ejecutándose, si fuese necesario, sin depender de ninguna *Activity*.

Típicos *services* son aquellos que periódicamente se conectan a algún servidor para ver si ha cambiado su información aunque estemos viendo otra aplicación. Se ejecuta en segundo plano y no depende de la *Activity* que lo haya lanzado.

Content providers

Los *Content providers* (proveedores de contenido) proporcionan una capa de abstracción para acceder a datos almacenados por una aplicación de modo que pueden ser accesibles a otras aplicaciones sin necesidad de comprometer la seguridad del sistema de ficheros. Las aplicaciones pueden guardar su información en la base de datos *SQLite* que proporciona Android, en ficheros o en otro sistema de almacenamiento.

Fragment

Los *Fragments* (fragmentos) aparecen a partir de la versión 3.0 de Android, para solucionar el problema de las múltiples pantallas. Un *fragment* está formado por la unión de varias vistas para crear un bloque funcional en la interfaz de usuario. Una vez creados, podremos combinar uno o varios *fragments* dentro de una actividad, según el tamaño de la pantalla.

Intents

Además de los cinco componentes principales, existen otros también necesarios para acceder o activar los bloques anteriores. Es el caso de los *Intent* (Intentos). Una intención representa la voluntad de realizar alguna acción mediante mensajes asíncronos que son lanzados constantemente a lo largo del sistema para notificar diversos eventos. Por ejemplo lanzar una actividad, lanzar un servicio, enviar un anuncio broadcast, comunicarnos con un servicio, etc.

Views

Las vistas son los elementos que componen la interfaz de usuario de una aplicación. Por ejemplo un botón, una entrada de texto, una imagen, etc. Todas las vistas van a ser objetos descendientes de una clase *View*, y por tanto pueden ser definidas utilizando código Java. También es posible definir dichas vistas utilizando un fichero XML y dejar que el sistema cree los objetos por nosotros. (Este fichero se describirá en detalle en el siguiente apartado).

Layout

Un *layout* es un conjunto de vistas agrupadas de una determinada forma. También son objetos descendientes de la clase *View*. Igual que la vistas, pueden ser definidos en código, aunque la forma más habitual y sencilla de definirlos es utilizando código XML.

1.1.4 Ficheros y carpetas de un proyecto Android

Un proyecto de Android está formado básicamente por un descriptor de la aplicación (*AndroidManifest.xml*), el código en Java y una serie de ficheros con recursos. Cada elemento se almacena en una carpeta específica.

src: Carpeta que contiene el código fuente de la aplicación. Como podemos observar, los ficheros Java se almacenan en carpetas según el nombre de su paquete.

gen: Carpeta que contiene el código generado de forma automática por el SDK. Nunca hay que modificar de forma manual estos ficheros.

BuildConfig.java: Define la constante *DEBUG* para que desde Java podamos saber si la aplicación está en fase de desarrollo.

R.java: Define una clase que asocia los recursos de la aplicación con identificadores. De esta forma los recursos podrán ser accedidos desde Java.

Android x.x: Código JAR, la API de Android según la versión seleccionada para compilar.

Android Private Libraries: Se indican las librerías asociadas al proyecto. Existen dos

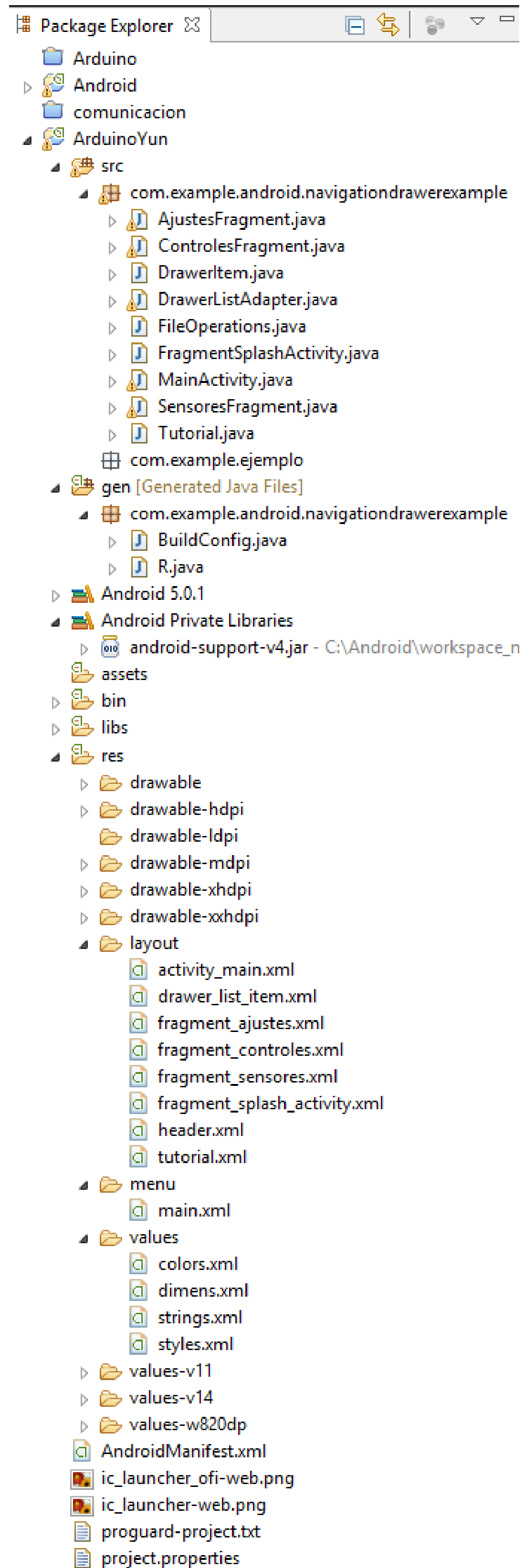


Ilustración 4: Estructura de un proyecto

alternativas para asociar una librería en Java. La primera consiste en crear un nuevo proyecto con el código y la segunda es utilizar un fichero .jar con el código ya compilado. La librería *android-support-v4* se añade automáticamente para permitir ciertas funcionalidades importantes no disponibles en el nivel de API mínimo seleccionado. Gracias a esta librería podemos utilizar elementos como Fragments, ViewPager o Navigation Drawer, que no están disponibles en el nivel de API mínimo seleccionado.

assets: Carpeta que puede contener una serie arbitraria de ficheros o subcarpetas que podrán ser utilizados por la aplicación (ficheros de datos, fuentes, etc.).

bin: En esta carpeta se compila el código y se genera el *.apk*, fichero comprimido que contiene la aplicación final lista para instalar.

libs: Código JAR con librerías que quieras usar en tu proyecto.

res: Carpeta que contiene los recursos usados por la aplicación.

drawable: En esta carpeta se almacena los ficheros de imágenes (JPG o PNG) y descriptores de imágenes en XML.

layout: Contiene ficheros XML con vistas de la aplicación. Las vistas nos permitirán configurar las diferentes pantallas que compondrán la interfaz de usuario de la aplicación.

menu: ficheros XML con los menús de cada actividad.

values: contiene ficheros XML como *styles.xml, strings.xml, colors.xml.* En ellos se definen parámetros fijos como estilos de textos o vistas de un objeto, mensajes de texto y caracteres y colores que definan a la aplicación.

AndroidManifest.xml: Este fichero describe la aplicación de Android. En él se indican las *actividades*, las *intenciones*, los *servicios* y los *proveedores de contenido* de la aplicación. También se declaran los permisos que requerirá la aplicación. Se indica la versión mínima y máxima de Android para poder ejecutarla, el paquete Java, la versión de la aplicación, etc. Se podría decir que este fichero es el que da vida a la aplicación.

ic_launcher_ofi-web.png: Icono de la aplicación de gran tamaño para ser usado en páginas web.

proguard-project.txt: Fichero de configuración de la herramienta ProGuard.

project.properties: Fichero generado automáticamente por el SDK.

1.1.5 Creación de un proyecto Android en Eclipse

En este apartado mostraremos como crear una aplicación par Android desde cero con el *IDE* Eclipse.

Una vez iniciado el programa, seleccionamos *File > New > Android Application Project.*

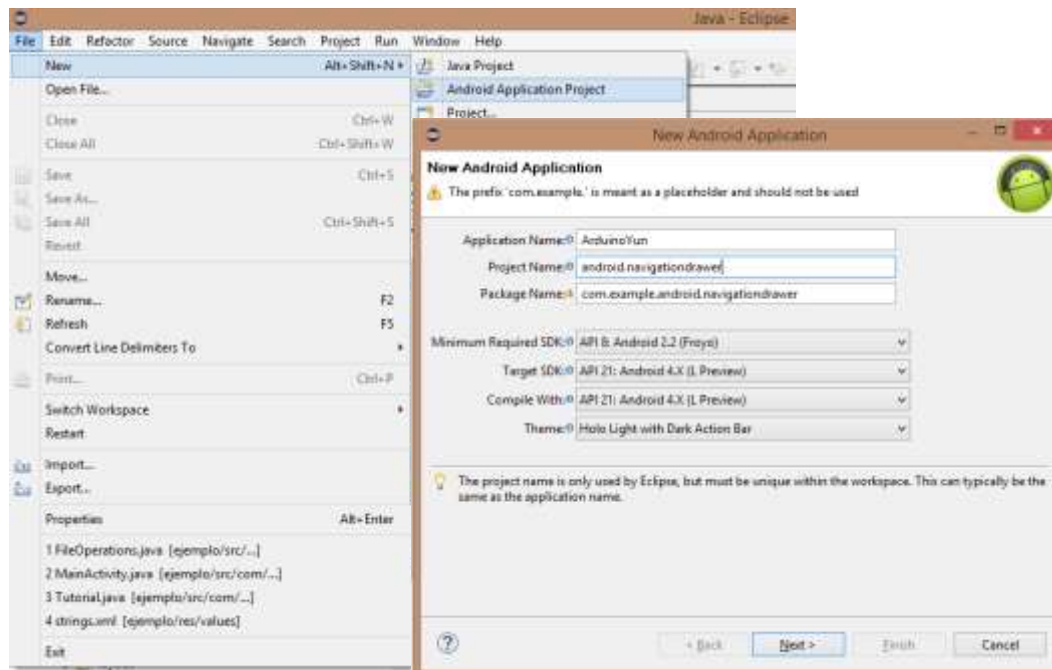


Ilustración 5: Configuración inicial Android Application Project.

A continuación, rellenemos los datos de la ventana emergente.

Application Name: Nombre de la aplicación que aparecerá en el dispositivo.
Project Name: Nombre del proyecto. Carpeta que contendrá los ficheros.
Package Name: Nombre del paquete único, instalado en el sistema.
Minimun required SDK: Valor mínimo del nivel de la API.
Target SDK: Versión más alta de Android que puede ser implementada.
Compile With: Versión de la plataforma con la que se compila la aplicación.
Theme: Estilo que utiliza la aplicación.

Después de pulsar "*Next*", nos encontramos con una nueva ventana. Aquí marcamos las dos primeras casillas para definir el icono que tendrá la aplicación y crear la actividad principal respectivamente. La cuarta casilla nos indica dónde se almacenará el proyecto. La ilustración 6 nos enseña cómo hacerlo.

En la ilustración 7, definimos el estilo y tamaño de nuestro icono.

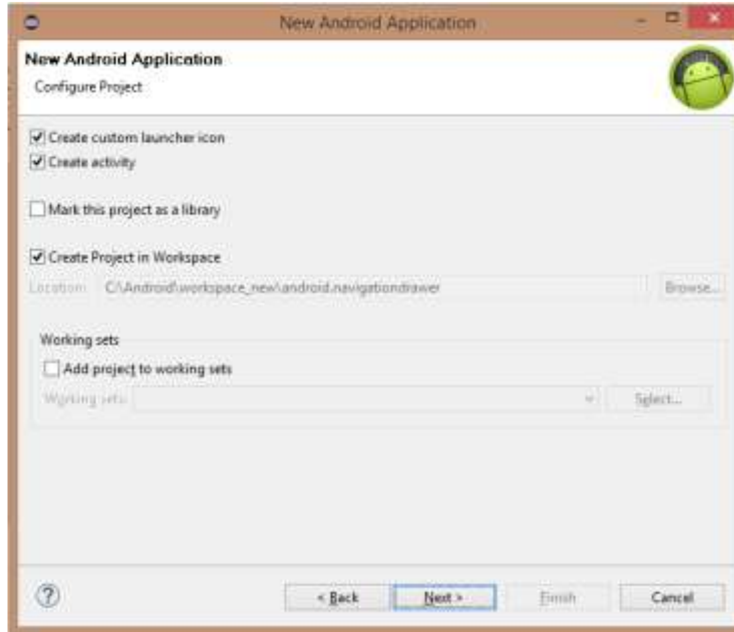


Ilustración 6: Definición de valores de un proyecto en Android.

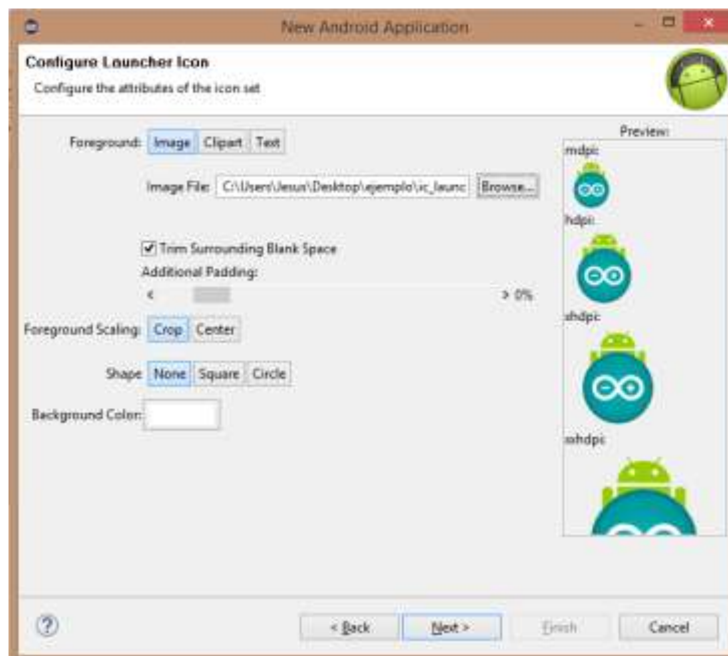


Ilustración 7: Creación y diseño del icono de la aplicación

Posteriormente, accedemos a la creación de la actividad (casilla que marcamos en la ilustración 6). En esta ventana, configuramos su tipo. Esta característica será por la que se defina el cuerpo central de la aplicación. En nuestro caso, como se explicó en el apartado *1.1.2 Base inicial de la aplicación,* implementaremos el menú de navegación para que se muestre en

la barra de acciones y la creación de un *fragment,* que nos permitirá una mayor fluidez entre las vistas y una menor carga de memoria para el sistema.

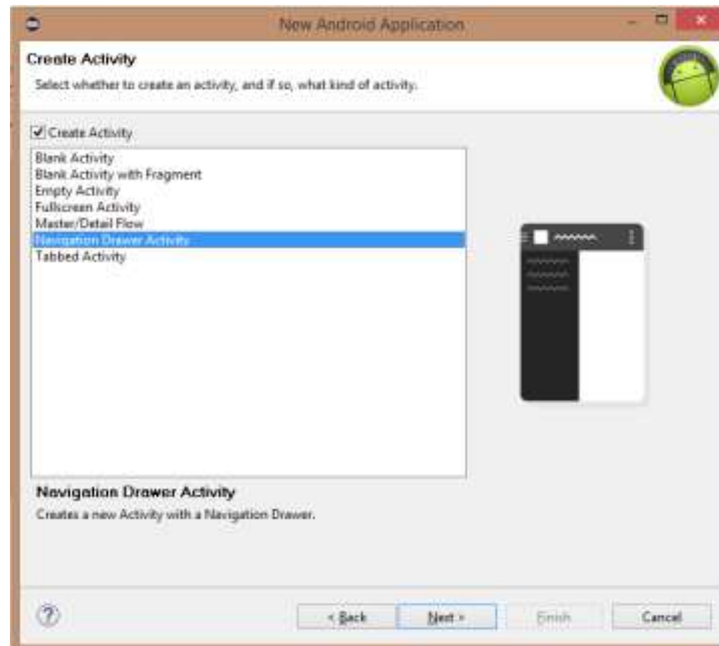


Ilustración 8: Creación de una actividad

El último paso para completar la configuración es definir los nombres que tomarán la activity, el *layout* asociado a ella, el *fragment* y el menú lateral o *Navigation Drawer*. (En el *Capítulo 3 Programación*, se explicará el uso de la *activity* en un diseño con *fragments.)*.

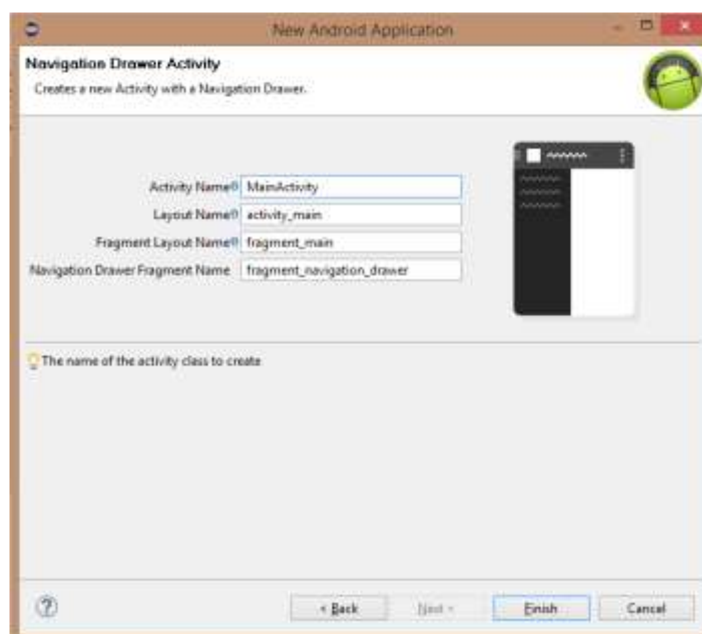


Ilustración 9: Nombres de la activity, layout, fragment y NavigationDrawer.

1.2 Arduino

Como ya vimos en el estudio previo, Arduino es una plataforma de realización de prototipos electrónicos compuesta tanto de hardware como de software. En este primer capítulo describiremos el segundo.

1.2.1 Entorno de desarrollo de Arduino

Como hemos comentado, Arduino es como un pequeño ordenador al que se le tiene que programar para poder interactuar con el medio a través de sus entradas y salidas.

Para poder programarlo se dispone de un IDE que nos facilitará en gran medida el proceso de desarrollo del código. Este entorno se puede descargar de manera gratuita a través de la página web *http://arduino.cc/en/Main/Software.* En el momento de la realización de este proyecto la versión descargada fue la 1.6.1 compatible con la tarjeta Arduino Yun, la cual utilizaremos en líneas posteriores.

En el anexo B, se detallan los pasos necesarios para instalar el software y todos los requisitos necesarios para el funcionamiento de Arduino.

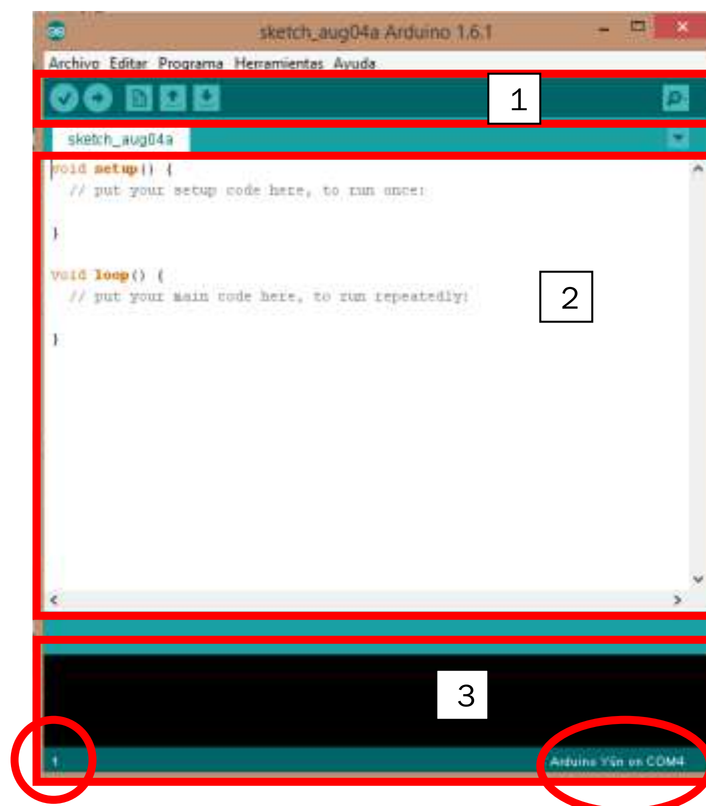


Ilustración 10: Entorno de desarrollo Android

Una vez instalado, para ejecutar el entorno de programación, habrá que seleccionar el fichero ejecutable *arduino.exe*. En la ilustración 10 observamos la ventana del entorno de desarrollo, la cual se divide en tres partes horizontales

En la parte superior tenemos unos botones con las acciones más comunes. La parte central será donde realizaremos la programación, donde se encontrará el código fuente también llamado *sketch*. Por último, la parte inferior, corresponde a la salida de la consola, donde podemos ver los errores y mensajes de información durante el proceso de compilación y envío del código a la placa Arduino. Debajo de esta consola encontramos una barra de información. En la parte izquierda se muestra el número de línea en la que está posicionado el cursor y a la derecha el modelo de la placa Arduino y el puerto en el que se encuentra conectada.

Ilustración 11: Botonera del IDE Arduino 1.6.1

De izquierda a derecha, los botones según la ilustración 11 sirven para:

Verificar: Verifica la sintaxis del código fuente que están cargando.
Cargar: Transmite el programa a la placa Arduino.
Nuevo: Crea un nuevo *sketch*.
Abrir: Muestra una selección de *sketch* correspondientes a ejemplos.
Guardar: Guarda el trabajo realizado con la extensión *.ino*
Monitor serie: Herramienta muy importante a la hora de depurar el programa. El monitor muestra los datos enviados por el puerto serie o USB desde Arduino y también permite el envío de datos hacia la placa.

La ilustración 12 muestra el monitor serie. Un parámetro fundamental es la configuración de la velocidad de comunicación con la placa Arduino medido en baudios (*baud*). El *baud*, es el número de cambios del estado de los bits por segundo, es decir, 9600 *baud* significa que en cada segundo se transmitirán 9600 caracteres.

Por defecto, los sketch no envían ningún tipo de dato al monitor serie, sino que debe hacerse mediante programación, indicando qué, cómo y cuándo enviarlo. Del mismo modo, tampoco se recibe ningún dato si no se especifica lo contrario mediante código.

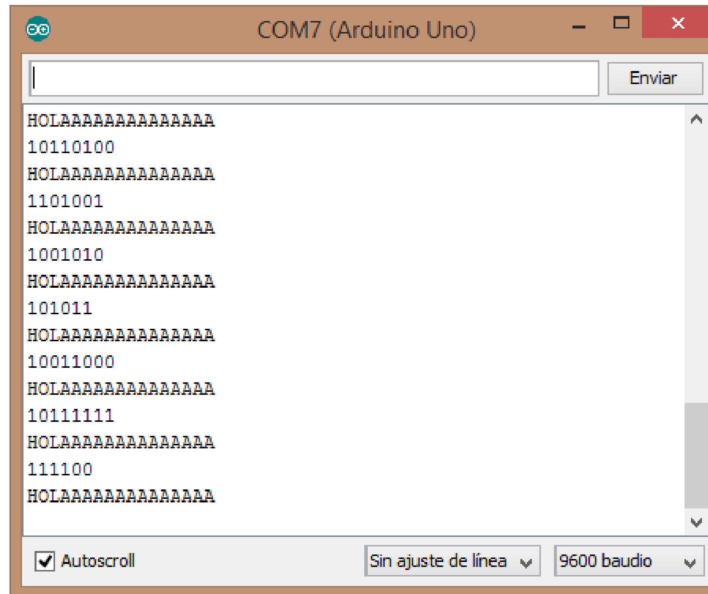


Ilustración 12: Monitor serie

1.2.2 Configuración del IDE

Para poder subir el código a la placa Arduino es necesario realizar unos simples pasos para establecer la conexión entre el microcontrolador y el ordenador.

En primer lugar, seleccionamos *Herramientas > Placa > Arduino Yún* Después, en el mismo apartado, *Herramientas > Puerto > COM X* donde *x* será el número de puerto al que se ha conectado Arduino al ordenador.

En el anexo B, se explica cómo consultar el COM desde el administrador de dispositivos del sistema.

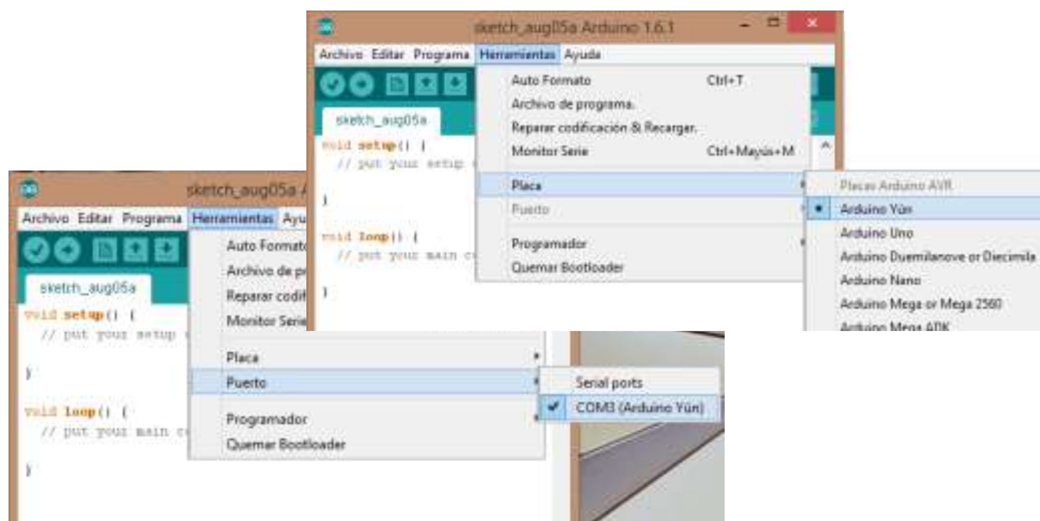


Ilustración 13: Configuración de la placa y puerto en el IDE de Arduino

1.2.3 Estructura de un programa

La programación de Arduino está basada en C y por lo tanto su estructura y funciones van acorde a este lenguaje.

Los bloques de código se encuentran entre llaves, que sirven para marcar el alcance de cada uno de ellos. Estos bloques pueden identificar funciones, bucles, condiciones, etc. Cada línea de código debe terminar con el carácter ";"

La estructura principal consta, además de la importación de librerías (no siempre necesaria) y la definición de variables, de dos bloques.

El bloque *setup()* es en realidad una función que se ejecuta cada vez que se enciende la placa o se pulsa el botón *reset* de la misma. Se ejecuta una sola vez por cada encendido y nos sirve para configurar el uso de pines, inicializar variables, etc.

```
void setup() {
  // put your setup code here, to run once:

}
```

Ilustración 14: Bloque setup() de un sketch

El bloque *loop()* es el cuerpo central del programa. Se ejecuta de forma repetitiva, después del *setup()*, y es el que tiene el control sobre la placa Arduino. Dentro de él es donde leemos las entradas, procesamos los datos y escribimos las salidas.

```
void loop() {
  // put your main code here, to run repeatedly:

}
```

Ilustración 15: Bloque loop() de un sketch

Por otra parte, se pueden definir más funciones fuera de estos bloques. Estas funciones pueden ser llamadas desde cualquier parte del programa principal (*recordemos el bloque loop()*).

En el *capítulo 3 Programación*, explicaremos todas las funciones, instrucciones y declaraciones necesarias para programar Arduino.

Capítulo 2
Descripción del hardware

En este segundo capítulo, explicaremos el hardware de Arduino. El estudio previo de sus características y funcionalidades nos servirán para sacar el máximo rendimiento a la placa y utilizarlo de una manera óptima en nuestro proyecto. Desde su microprocesador hasta sus entradas y salidas.

2.1 Arduino Yun

Arduino Yun es uno de los modelos fabricados por Arduino y que utilizaremos este proyecto para establecer la conexión con Android.

Esta placa, como característica principal, combina dos procesadores. El primero de ellos, el Atmel ATmega 32U4, que se encarga de ejecutar las aplicaciones de Arduino. El segundo, el Atheros AR9331, compatible con el servidor Linux, se encarga de la gestión de redes (Ethernet, WiFi), el USB Host (pendrive, teclados, ratones) y la microSD (almacenamiento de datos).

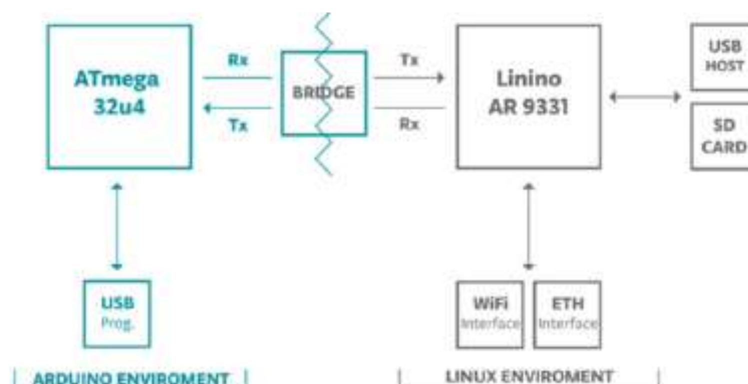


Ilustración 16: Comunicación entre Arduino y Linino (Linux)

Esta placa dispone de una combinación de 20 entradas y salidas digitales con 7 canales *PWM* y 12 canales de entrada analógicas. La tensión de funcionamiento es de 5 Voltios y no dispone de regulador de tensión como si cuentan otros modelos de Arduino. De esa forma, en ningún caso se debe superar dicha tensión, ya que produciría una sobrecarga y dañaría el microcontrolador.

Otras de las características de Arduino Yun se describen en las siguientes tablas.

AVR Arduino	
Microcontrolador	ATmega32U4
Tensión de funcionamiento	5V
Voltaje de entrada	5V
Corriente DC por E/S Pin	40 mA
Corriente DC de 3.3V Pin	50 mA
Memoria flash	32 KB
SRAM	2,5 KB
EEPROM	1 KB
Velocidad de reloj	16 MHz

Tabla 2: Características del entorno Arduino

Linux microprocesador	
Procesador	Atheros AR9331
Arquitectura	MIPS@400MHz
Tensión de funcionamiento	3,3V
Ethernet	IEEE 802.3.10/100Mbit/s
WiFi	IEEE 802.11b/g/n
USB Tipo-A	2.0 Host
Lectura de tarjetas	MicroSD
RAM	64 MB DDR2
Memoria flash	16 MB

Tabla 3: Características del microprocesador Linux

Alimentación

Como se detallan en las características anteriores, debe ser alimentado exclusivamente con 5V en corriente continua. Esta alimentación podemos obtenerla, bien a través de la conexión micro-USB que incorpora nuestra placa, o bien a través del pin "*Vin*", con un regulador de tensión que ofrezca la tensión citada anteriormente.

Entradas y salidas

Los pines digitales, tanto en las entradas como en las salidas, trabajan a 5 Voltios, pudiendo recibir un máximo de 40 mA (*miliamperios*). Tiene una resistencia pull-up de 20 a 50 kOhm desconectada por defecto. Las funciones asociados a estos pines son por ejemplo *pinMode()*, *digitalWrite(), digitalRead()* entre otras, las cuales detallaremos en el siguiente capítulo.

Como detallamos en la ilustración 17, la función de cada pin es la siguiente:

Serial: 0 (RX) y 1 (TX). Se utiliza para recibir (RX) y transmitir (TX) datos serie TTL.

TWI: 2 (SDA) y 3 (SCL). Soporta la comunicación TWI.

Interrupciones externas: 3, 2, 0, 1 y 7. Estos pines se pueden configurar para provocar una interrupción.

PWM: 3, 5, 6, 9, 10, 11 y 13: Proporciona una salida PWM de 8 bits con la función *analogWrite()*.

SPI: En el conector ICSP. Estos terminales soportan la comunicación SPI a través de la librería SPI

LED 13: El pin digital 13 lleva conectado un LED integrado en la propia placa (L13). Se encenderá cuando dicho pin se configura como salida y adopte un valor *HIGH*, mientras que con el valor *LOW* se apaga.

Otros indicadores de estado: Como vemos en la ilustración 18, el Yun incorpora otros leds para ver el estado de la alimentación, la conexión WLAN, conexión WAN y USB.

AREF: Voltaje de referencia para las entradas analógicas. Se utiliza *analogReference()*.

Entradas analógicas: A0 - A5, A6 – A11 (en los pines digitales 4, 6, 8, 9, 10 y 12). El Arduino Yun tiene 12 entradas analógicas. Cada entrada proporciona 10 bits de resolución (1024 valores).

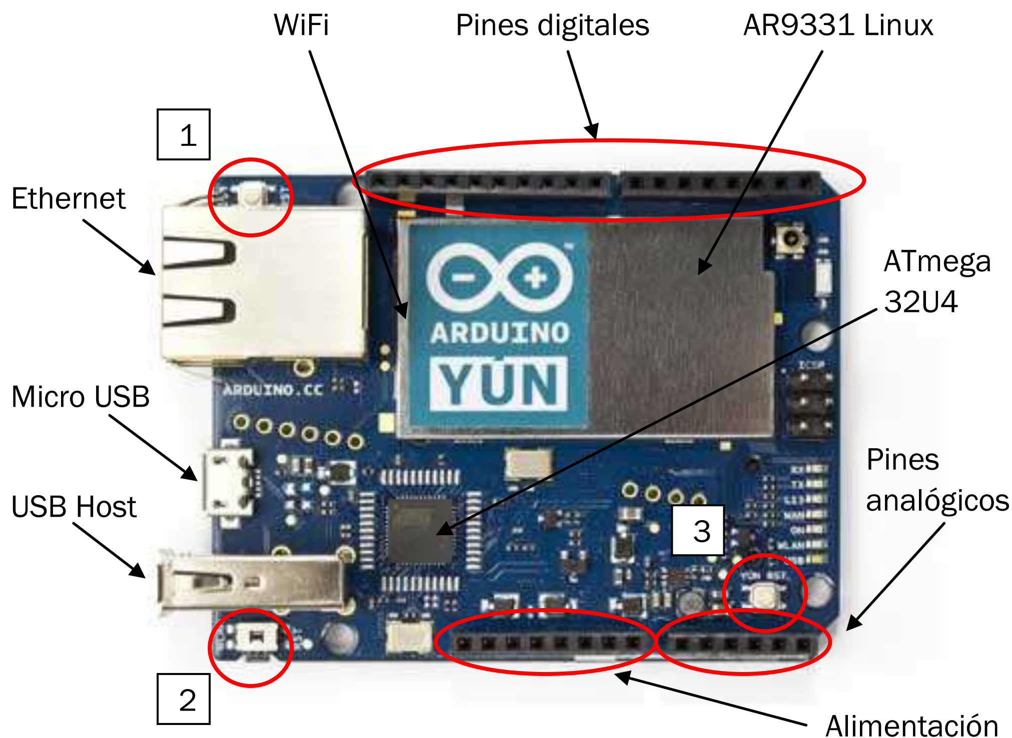


Ilustración 17: Placa Arduino Yun

Como vemos en la ilustración 17, existen tres botones en la placa Arduino con una función diferente cada uno:

1. ***Botón RST 32U4***: Inicializa el ATmega 32U4 e inicializa todos los programas de Arduino. No se pierden los programas ya que se encuentran en la memoria flash.
2. ***Botón RST WLAN***: Tiene una doble función. En primer lugar, pulsando 5 segundos sobre el botón, conseguimos restaurar el WiFi a la configuración de fábrica. La segunda función, presionando durante 30 segundos, conseguimos restaurar la imagen del sistema Linux, con la consiguiente pérdida de aplicaciones instaladas en Linux.
3. ***Botón RST Yun***: Inicializa el AR9331 y el sistema Linux. Todos los datos almacenados en la memoria RAM se perderán y todos los programas que se estén ejecutando se terminarán.

2.2 Configuración inicial del Arduino Yun

En este apartado configuraremos paso a paso nuestro Arduino Yun para establecer una conexión con nuestro ordenador.

En primer lugar conectaremos la placa a nuestro ordenador a través del Micro USB del primero y el USB del segundo. Debemos tener en cuenta que el Arduino Yun tiene un comportamiento similar a un router WiFi, por lo que cuando lo conectemos, no tendremos la señal disponible de inmediato, sino que será necesario esperar al menos un minuto hasta que habilite la red inalámbrica.

Una vez se haya habilitado, nos vamos a las redes inalámbricas de Windows, donde veremos disponible una red llamada ArduinYún-XXXXXXXXXX.

Ilustración 18: Red inalámbrica generada por Arduino Yun

El siguiente paso será conectarnos a esa red. Abriremos el navegador y accederemos tecleando en la barra de URL la dirección *arduino.local* o *http://192.168.240.1*

Una vez accedido, introduciremos la contraseña, que por defecto es "arduino". La siguiente ventana, como vemos en la ilustración 19, muestra la configuración del Arduino Yun. Introduciremos un nombre y una contraseña para acceder a la placa.

En esta configuración, será necesario establecer una red de Internet vía WiFi desde la cual el Arduino Yun se conectará a la web. Completaremos otros parámetros como el nombre que daremos a nuestro Arduino y la contraseña. En el siguiente campo seleccionaremos nuestra zona horaria.

A continuación, los parámetros inalámbricos. Seleccionaremos la red WiFi con la que queremos conectar nuestro Arduino y el SSID de la red a la

que conectarse, además de introducir la contraseña propia. Pulsaremos el botón de configurar y después de un reinicio de la placa, la tendremos configurada. Ver ilustración 20.

Ahora el Arduino Yun estará conectado a la conexión WiFi requerida y tanto el dispositivo como el ordenador estarán conectados en la misma red.

Ilustración 19: Configuración inicial Arduino Yun

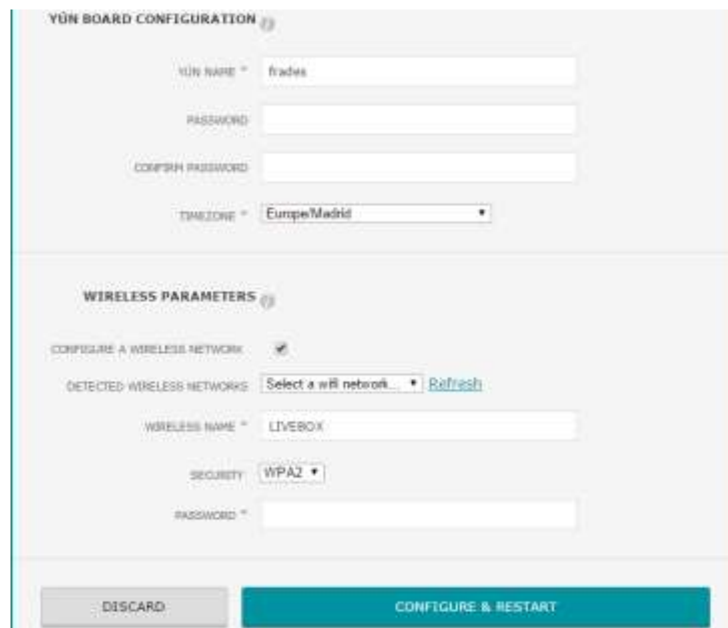


Ilustración 20: Parámetros de configuración de Arduino Yun

2.3 Datos analógicos y digitales

Para utilizar Arduino, es necesario conocer las diferencias entre datos analógicos y digitales. En este apartado vamos a introducirnos en ellas.

Cuando hablamos de entradas digitales, debemos pensar en que solo se pueden tener dos valores: encendido y apagado, 1 y 0, *on* y *off*, `true` y `false`, `HIGH` y `LOW` y otras maneras de nombrarlo. De forma general se toma que la representación de apagado son 0 voltios, mientras que la de encendido es de 5 voltios.

Debido a que en muchos casos es imposible llegar a estos valores extremos, se toma como referencia un valor y cuando el voltaje es mayor de ese valor, se tomará como encendido y cuando sea menor se tomará como apagado. Este valor de referencia en Arduino es de 3 voltios.

En cambio, si hablamos de entradas analógicas, los valores pueden ser múltiples. Cuando un Arduino recibe una entrada analógica, lo hará entre un voltaje de 0 y 5 voltios, que será transformado en los valores 0 y 1023 respectivamente. Esta conversión, se realiza multiplicando la entrada de voltaje en ese instante por 1023 divido entre 5 voltios (por ejemplo: 2.3*1023/5V=471). Como resultado obtenemos uno de los 1024 valores comprendidos entre 0 y 1023.

En cuanto a las salidas digitales, idénticas a las entradas, obtenemos valores *todo o nada*. Simplemente, tendremos actuadores en modo encendido o apagado.

Para el caso de las salidas analógicas, identificadas con el símbolo (~) junto a su pin, soportan el uso de modulación por amplitud de pulso o PWM. Al utilizarlo, enviamos a la salida pulsos de 0 y 1 muy rápido y dependiendo de la duración del pulso en 1 significarán cantidades distintas, simulando valores entre 0 y 5 voltios. Los valores de escritura en la salida analógica PWM, van de 0 a 255.

Capítulo 3
Programación

La programación, será la protagonista en este capítulo. Hablaremos de cada bloque de instrucciones y sus funciones. Estará dividido principalmente en dos apartados. Uno para describir la programación de Android y el segundo la de Arduino.

3.1 Diseño de la interfaz de usuario: *Vistas* y *Layouts*

El primer paso para realizar una aplicación es realizar el diseño gráfico de la interfaz. Como ya vimos en el capítulo 1, hicimos una rápida y breve descripción de la interfaz consiguiendo una idea que ahora en este capítulo desarrollaremos mediante la programación.

En Android, la interfaz de usuario principalmente se diseña utilizando un código marcado como XML, similar al HTML.

Los *layout* están compuestos de elementos que pueden ser contenedores *widgets*. Los contenedores son elementos visuales o no, que puede a su vez contener otros contenedores o *widgets*; son elementos que descienden de la clase *ViewGroup* que a su vez desciende de la clase *View*. Los *widgets* de interfaces gráficas se pueden encontrar dentro del paquete *android.widget* y son casi todos ellos elementos visuales que descienden de la clase *View*.

La manera en la que se mostrarán o cómo se comportarán los contenedores o los *widgets* viene dado por los atributos que se le den a cada elemento en su definición dentro del documento XML.

Como ya definimos anteriormente y para recordar estos términos que se usarán mucho en este capítulo, una *vista* es un objeto que se puede dibujar y se utiliza como un elemento en el diseño de la interfaz de usuario (un botón, una imagen, un texto…) Cada uno de estos elementos se define como una subclase de la clase *View*; por ejemplo, la subclase para representar un texto es un *TextView,* para una imagen es una *ImageView* y para un botón es un *Button.*

Para comenzar a trabajar sobre el fichero XML, accederemos a él a través de la ruta `res/layout/activity_main.xml`, que nos generó automáticamente el programa al crear nuestro primer proyecto *(Ver apartado 1.1.5 Creación de un proyecto Android en Eclipse).*

Seguidamente, como vemos en la ilustración 21 y 22, tenemos dos interfaces de desarrollo, una a través del código XML y otra de edición visual de vistas. Ambas son complementarias y mientras en una de ellas realizas la programación, en la otra pestaña se va completando con objetos y vistas. Para realizar el cambio entre ellas, existen dos botones en la parte inferior.

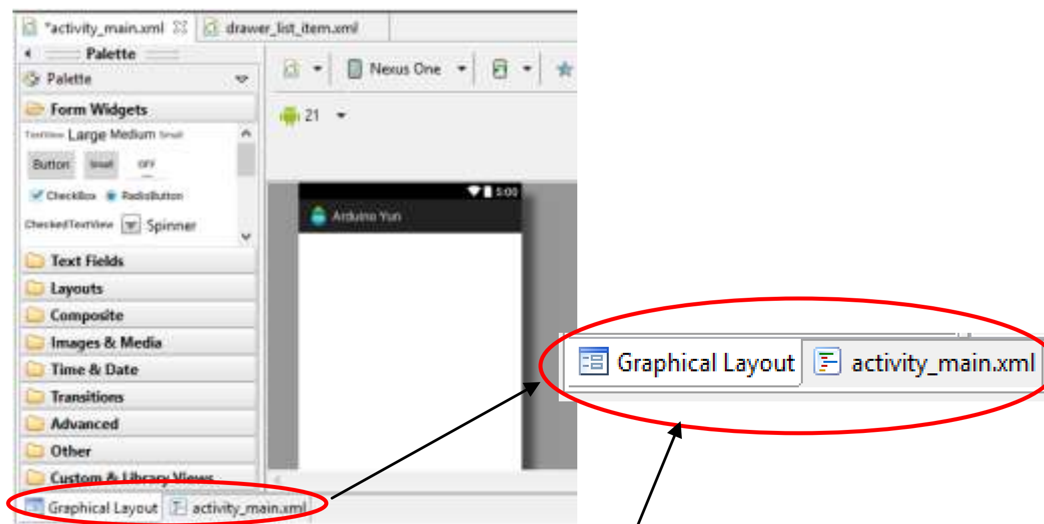


Ilustración 21: Interfaz de diseño. Graphical Layout

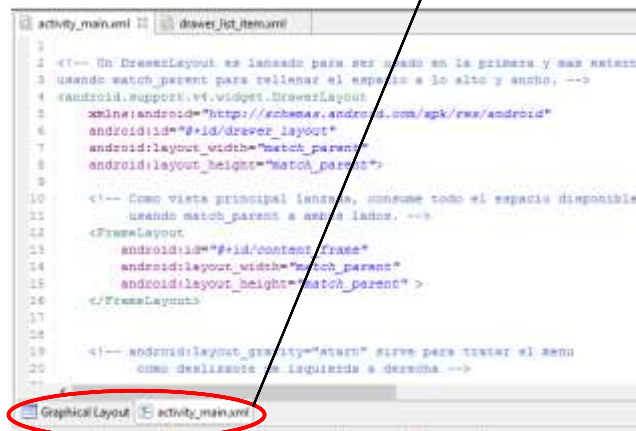


Ilustración 22: Interfaz de diseño. XML

El layout principal que tendrá la aplicación será `activity_main.xml`. En la edición de vistas no se aprecia nada, ya que como dijimos, pueden existir vistas ocultas.

Como ya conocemos el diseño y la estructura que queremos tener en nuestra aplicación implementaremos la vista *DrawerLayout*, necesaria para poder interactuar con el menú lateral desplegable.

DrawerLayout actúa como un contenedor de nivel superior en la vista, la cual permite interactuar con el "cajón" o menú lateral sacándolo desde el borde de la ventana. También podemos desplazar la barra de navegación fuera de la zona visible del layout y con un "click" o gesto de deslizamiento, volver a hacerla visible. Ésta es una de las grandes ventajas que nos ofrece, ya que no nos ocupa espacio a la hora de interactuar con la interfaz.

Para poder trabajar con esta vista, debemos añadir la librería `android.support.v4.widget.DrawerLayout` Esta versión *android.support.V4* nos permite trabajar con ciertas funcionalidades importantes no disponibles en el nivel de API mínimo seleccionado. Gracias a esta librería podemos utilizar elementos como *Fragments*, *ViewPager* o *Navigation Drawer*, que no están disponibles en el nivel de API mínimo seleccionado y así poder funcionar en sistemas operativos que no tengan sus dispositivos actualizados.

```
<!-- Un DrawerLayout es lanzado para ser usado en la primera
y mas externa capa de la vista  usando match_parent para rellenar
el espacio a lo alto y ancho. -->
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

FrameLayout permite el cambio dinámico de los elementos que contiene. Esta característica se puede usar en actividades pero con la aparición de los fragmentos tiene más sentido. *FrameLayout* posiciona las vistas usando todo el contendor, sin distribuirlas espacialmente. Se comporta como una pila de elementos, según se van añadiendo elementos a él, se van colocando uno encima de otro, quedando ocultos los que están en la parte inferior. Para gestionar el uso entre ellos utilizaremos la clase *gettFragmetManager*.

Ahora entendemos por qué la vista padre es el DrawerLayout, quien va a estar presente en todas ellas y a partir de ahí, las clases hijas, como son el FrameLayout y el ListView. De esta forma, siempre tendremos la posibilidad

de mostrar el menú desplegable, aún añadiendo encima de ella nuevas vistas. (Una clase es descendiente de otra cuando se usa la tabulación).

```
<!-- Como vista principal lanzada, consume todo el espacio
  disponible usando match_parent a ambos lados. -->
<FrameLayout
    android:id="@+id/content_frame"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
</FrameLayout>
```

La siguiente clase *View,* corresponde a un *ListView*, con el cuál visualizamos una lista de varios elementos. Para definirla, es necesario crear otro *Layout* que contenga los detalles a mostrar por la lista.

```
<!-- android:layout_gravity="start" sirve para tratar el menu
      como deslizante de izquierda a derecha -->
  <ListView
        android:id="@+id/left_drawer"
        android:layout_width="250dp"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:background="@color/LightGrey"
        android:choiceMode="singleChoice"
        android:divider="@android:color/transparent"
        android:dividerHeight="0dp" />
</android.support.v4.widget.DrawerLayout>
```

Cada clase debe tener un identificador, que viene dado por `“@+id/nombre”`, seguido del tamaño de ancho y alto que debe ocupar el elemento o vista, identificado con `android:layout`. Después, como en el caso del *ListView*, podemos añadir declaraciones como `android:background` que muestra el fondo de la vista de un color determinado. Algunos de los recursos como textos o colores, se añaden al fichero `res/values/string.xml`

Como hemos mencionado en el caso del *ListView,* es necesario crear un nuevo *layout,* `drawer_list_item.xml`, con el que detallaremos la lista y sus componentes. Como clase padre tendremos un *RelativeLayout* y descendiendo como clases hijas un *TextView* y un *ImageView.*

RelativeLayout dispone los elementos con relación a otro o al padre. Permite comenzar a situar los elementos en cualquiera de los cuatro lados del contenedor. La posición de los elementos se calcula de manera relativa a su elemento padre o a la posición de los elementos hermanos.

Con ***TextView*** mostramos un texto y opcionalmente nos permite su edición. Otros parámetros para su visualización son el tamaño de letra, la posición, la distancia con otros objetos y el tamaño del TextView en sí. En este caso mostramos un texto que encontraremos en una clase Java de vectores para mostrar por orden los textos en la lista.

ImageView muestra una imagen arbitraria. Esta se encuentra en una clase Java creada desde el código principal para invocar la llamada y mostrar por orden en un vector, las imágenes de la lista del menú de navegación desplegable.

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:padding="5dp">

    <TextView
        android:id="@+id/text1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:gravity="center_vertical"
android:minHeight="?android:attr/listPreferredItemHeightSmall"
        android:paddingLeft="70dp"
        android:paddingRight="30dp"
android:textAppearance="?android:attr/textAppearanceMedium"
        android:textColor="@color/Black" />

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/icon"
        android:layout_alignParentLeft="true"
        android:layout_marginRight="16dp"
        android:gravity="center_vertical" />
</RelativeLayout>
```

El tercer layout creado, `fragment_ajustes.xml`, es ya uno de los fragmentos con los que trataremos en el código Java más adelante. En este caso, en la edición de vistas tenemos objetos de la clase *View* tales como textos, imágenes, botones y spinners. Los dos primeros ya explicados.

La subclase ***Button*** representa un botón que puede ser pulsado. Deberá tener escuchadores que indiquen una acción determinada cada vez que el botón se presione. Como estamos solamente en la parte del diseño, no entraremos en más detalle que el visual. En la ilustración 23 se muestran los objetos que componen este tercer layout.

En cuanto al ***Spinner***, es una vista cuyos hijos están determinados por un *adapter* que viene definido en el código Java. Proporciona de una manera rápida un conjunto de selecciones en una lista oculta desplegable.

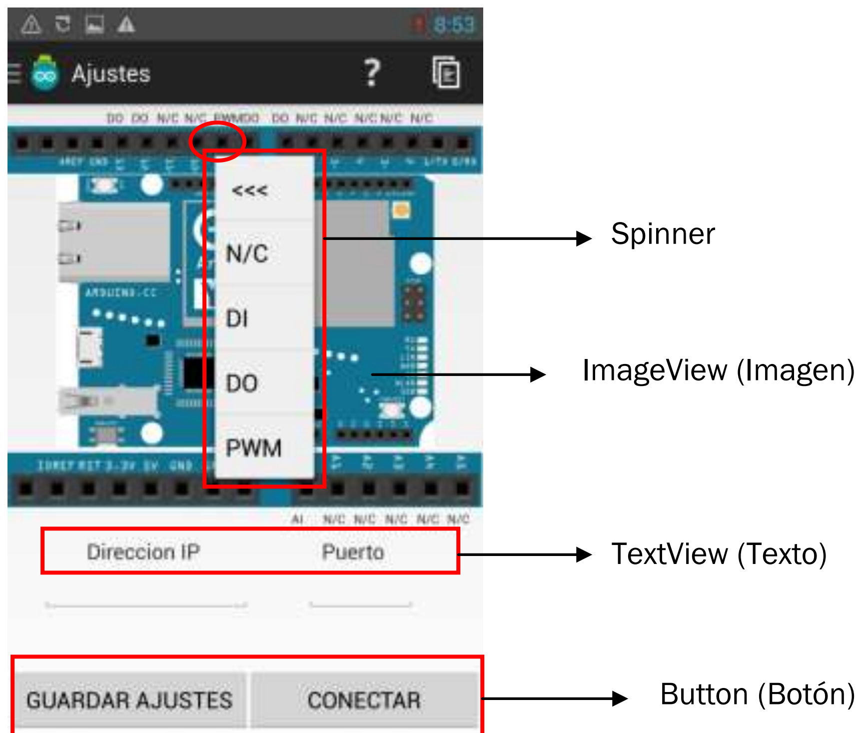


Ilustración 23: Vista del Layout "Ajustes"

Otro de los layout generados para el control de la aplicación es `fragment_controles.xml`, el cuál incorpora vista como textos, botones, interruptores y barras de desplazamiento, checkboxs y un scrollview.

Switch, interruptor de dos estados. Podemos cambiar su posición deslizando con el dedo o pulsando directamente sobre el nuevo estado. Podemos definir características como el texto que le define, `android:text`, sus dimensiones `android:layout` y el nombre al que realizaremos la llamada cuando sea invocado el botón al ser pulsado desde el código Java (`android:onClick`). Esta declaración puede hacerse o no, dependiendo del código que se implemente.

```
<Switch
        android:id="@+id/switch1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="click2"
        android:text="DO2" />
```

Checkbox, casilla de verificación que permite al usuario seleccionar una opción. En nuestra vista, está desactivada, ya que la función que realiza no puede verse alterada por el usuario y por lo tanto solo indica si ha sido seleccionada (✓) o no lo ha sido (vacio). Sigue las mismas declaraciones que muchos de los otros widgets. Como novedad, la declaración `android:enabled` que permite por defecto al inicio de lanzar el layout, habilitar o no, la vista para modificarla.

```
<CheckBox
      android:id="@+id/CheckBox1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginLeft="0dp"
      android:enabled="false"
      android:text="DI2" />
```

Seekbar, barra de extensión deslizable. El usuario puede alcanzar el destino deslizando con el dedo el círculo a través de la barra o pulsando directamente sobre la línea de progreso. Una característica llamativa de esta vista es el poder ajustar el paso de la barra, (no confundir con la longitud) acorde a un número y limitarla mediante la declaración `android:max`

```
<SeekBar
      android:id="@+id/seekBar1"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_marginLeft="45dp"
      android:layout_marginRight="25dp"
      android:max="255" />
```

ScrollView, diseño contenedor basado en una jerarquía en la que el usuario se puede desplazar verticalmente a través de la interfaz, permitiendo mostrar más objetos y vistas de la que puede mostrar la pantalla física. La ilustración 24 nos enseña los componentes que contiene el layout.

```
<ScrollView
      android:id="@+id/scrollView1"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:layout_marginBottom="50dp" >
</ScrollView>
```

Las cuatro primeras declaraciones se pueden repetir atendiendo al número de pines que tengamos para tener el mismo formato, variando en

algunos layouts, aquellos en los que no se dispongan de salida analógica PWM. En el anexo C se muestran los códigos tanto de los programas de Android como de Arduino.

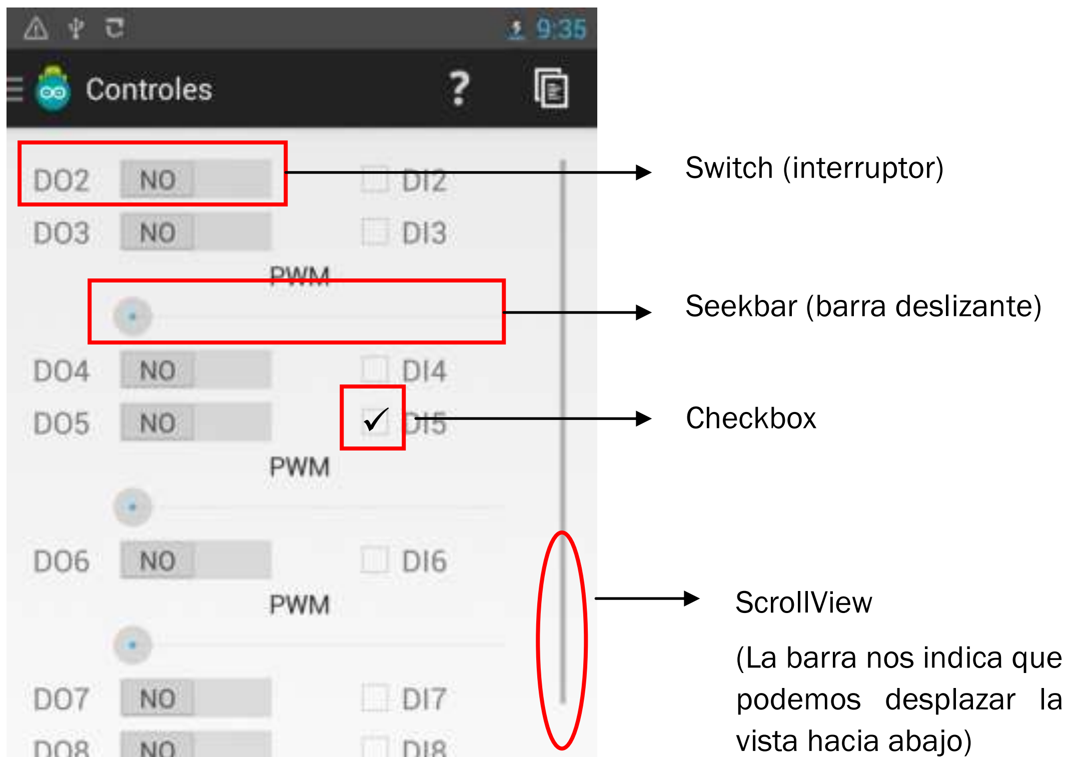


Ilustración 24: Vista layout controles

El siguiente layout que explicamos es el creado para controlar la parte de los sensores de Arduino, `fragment_sensores.xml`. Este layout incluye progressbar, textos y un botón.

Progressbar, elemento gráfico de control que visualiza el progreso de un estado u operación. Esta vista no puede ser modificada por el usuario.

```
<ProgressBar
                android:id="@+id/progressBar1"
                style="?android:attr/progressBarStyleHorizontal"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:max="1024"
                android:visibility="visible" />
```

Los siguientes layouts creados, han sido de apoyo para mejorar el diseño. En primer lugar, la cabecera del menú lateral desplegable, diseñada con una imagen representativa del proyecto a quien está dedicado. La visualización de la imagen se muestra cada vez que se abre el menú y está

situada como cabecera de la lista. El fichero lo podemos encontrar bajo el nombre de `header.xml`.

El layout `tutorial.xml`, es una pequeña ayuda que ofrecemos al usuario para iniciarle en el manejo de la aplicación. Como característica principal, se ha diseñado una vista estática y mediante el código Java hemos ido ocultando y mostrando vistas según avanzamos entre las distintas ventanas de "ayuda". De esta forma, conseguimos una única actividad y evitamos que por cada paso de página carguemos nuevas actividades con el consecuente cargo de memoria al estar apilándose nuevas vistas.

Por último, fragment_splah_activity.xml nos muestra una imagen al iniciar la aplicación de unos pocos segundos de duración. Actualmente se realiza esta técnica como firma de la aplicación por el autor.

La ilustración 25 nos enseña el diseño creado por la edición visual de la pestaña de la interfaz "*Graphical Layout*" de los tres últimos layouts.

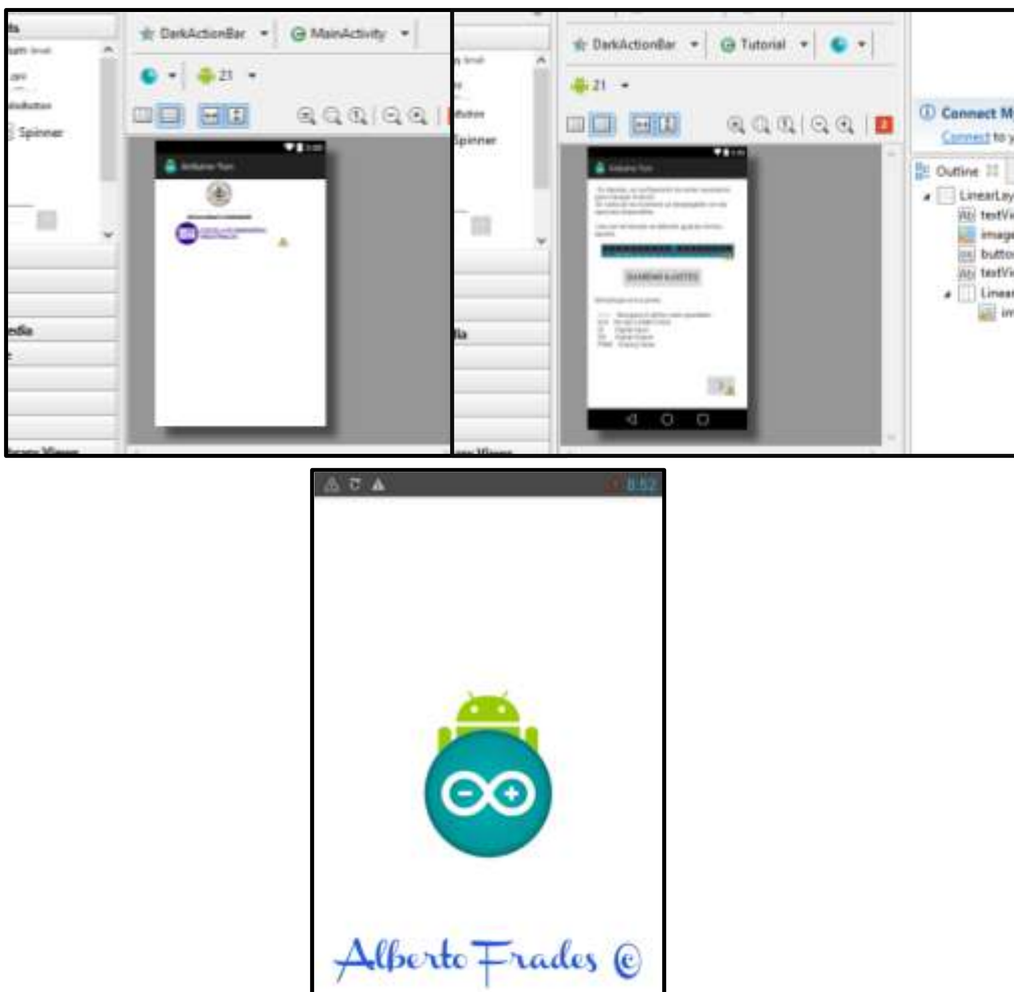
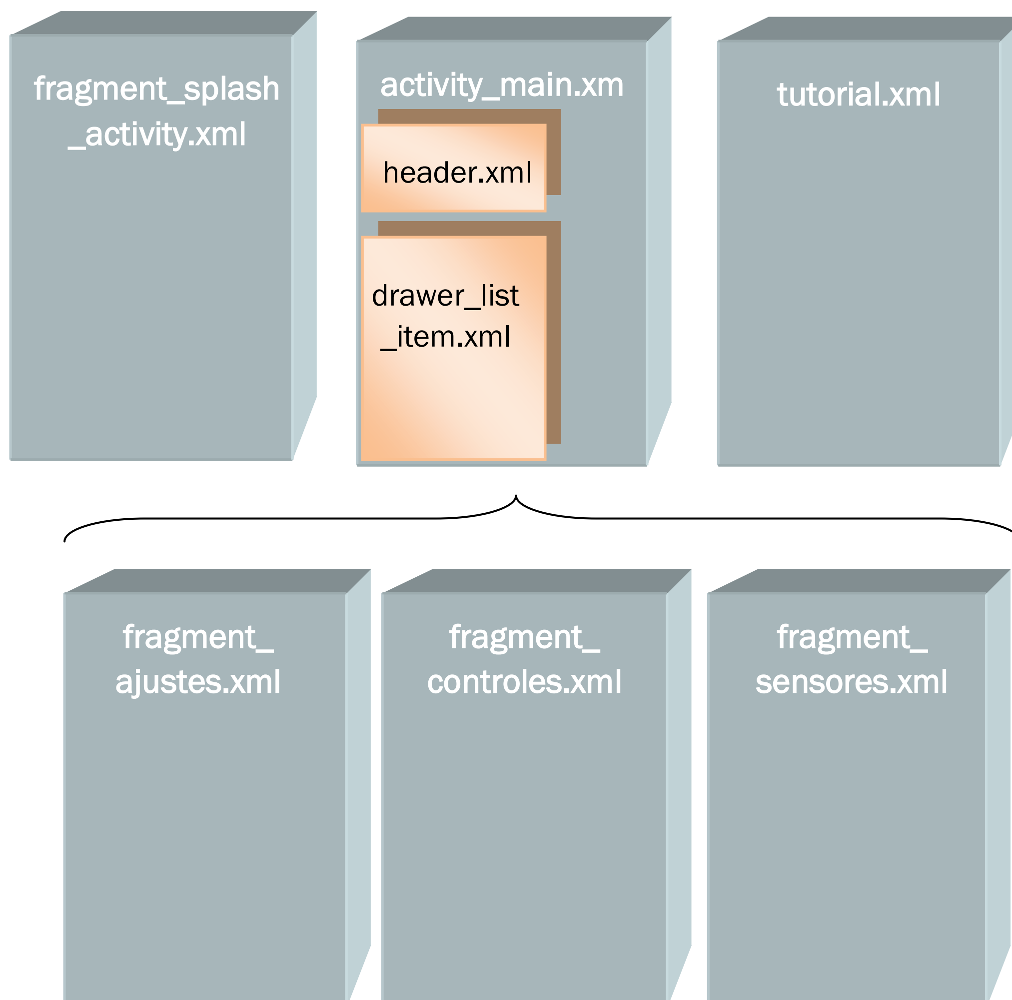


Ilustración 25: Layouts header.xml, tutorial.xml y splash.xml

En resumen, tenemos dos *activity*, cuatro *fragments*, y dos layouts más que complementan a la actividad principal `activity_main.xml`. La ilustración 26 muestra un esquema general de la composición de las vistas y la jerarquía.

Como hemos dicho, la actividad principal de la aplicación es `activity_main.xml`. A través de ella gestionamos el uso del menú lateral, el cual contiene la lista de opciones (`drawer_list_item.xml`) y la cabecera (`header.xml`). Además, los cuatro *fragments* (`fragment_ajustes.xml`, `fragment_controles.xml`, `fragment_sensores.xml` y `fragment_splash_activity.xml`) se comunican entre sí por medio de la actividad principal a la hora de pasarse parámetros.

Por otra parte, el ultimo layout que nos queda (`tutorial.xml`), funciona como actividad independiente, abriendo una nueva ventana para mostrar el diseño de la vista.



3.2 Navigation Drawer o Menú de navegación

En este apartado explicamos la implementación y el uso del *Navigation Drawer* en nuestra aplicación para crear una navegación a través de un menú deslizante.

Como ya explicamos en apartados anteriores, esta característica nos muestra un panel deslizante cuyo objetivo es dotar al usuario con una navegación más cómoda entre las opciones más frecuente de la aplicación.

Existen dos formas para visualizar su contenido: La primera es deslizando desde el borde izquierdo de la pantalla hacia la derecha. La segunda, presionando el icono de la Action Bar (icono de la aplicación situado en la parte superior de la ventana).

Cada vez que el usuario seleccione una categoría desde el menú, el layout principal se actualizara para proyectar su respectivo contenido. Como ya conocemos del apartado anterior, este proyecto se compone de un *DrawerLayout* con un *RelativeLayout* para proyectar el contenido de la *ListView* (con su respectivo contenedor) para las opciones con un texto y una imagen (*TextView* e *ImageView* respectivamente).

Ya vimos en la página anterior una representación esquemática o jerárquica de los layouts. A partir de ello, sabemos que la actividad principal se complementa con otros dos layouts. Lo mismo ocurre en el código de Java, donde `Main_activity.java` invoca y llama a otras dos clases (`DrawerItem.java` y `DrawerListAdapter.java`) para que *Navigation Drawer* funcione en todas las vistas.

3.2.1 Adaptador para el ListView

Un adaptador es un puente de comunicación entre un ***AdapterView*** y los datos que queremos mostrar en una vista. De esta forma, dinámicamente se irá actualizando la interfaz con nuestros datos. El más utilizado es ***ArrayAdapter*** .

Estos datos se encuentran en el menú de navegación con una lista cuyos ítems poseen un icono por cada elemento y el texto que representa la opción. Se implementa una clase que represente el ítem de la lista para relacionar un recurso *drawable* y un *string* (imagen y cadena de caracteres respectivamente). Con el layout creado a partir de una imagen y un texto, extendemos la clase *ArrayAdapter* para relacionar los datos.

A continuación, vemos la clase representativa de cada elemento con su definición. La actividad principal cada vez que se invoque al menú lateral, llamará a esta clase `DraerItem.java` y recibirá los argumentos de texto e imagen correspondiente a cada opción.

```
public class DrawerItem {
      private String name;
      private int iconId;
    public DrawerItem(String name, int iconId) {
        this.name = name;
        this.iconId = iconId;}
    public String getName() {
        return name;}
    public void setName(String name) {
        this.name = name;}
    public int getIconId() {
        return iconId;}
    public void setIconId(int iconId) {
        this.iconId = iconId;}
}
```

Después, solo nos queda implementar el adaptador `DrawerListAdapter.java` usando como tipo de entrada los elementos de la clase anterior, `DrawerItem.java`

```
public class DrawerListAdapter extends ArrayAdapter {
      public DrawerListAdapter(Context context, List objects) {
            super(context, 0, objects);
      }
@Override
public View getView(int position, View convertView, ViewGroup
                  parent) {

      if(convertView == null){
            LayoutInflater inflater =
            (LayoutInflater)parent.getContext().
            getSystemService(Context.LAYOUT_INFLATER_SERVICE);
            convertView = inflater.inflate
            (R.layout.drawer_list_item, null);
      }
ImageView icon = (ImageView) convertView.findViewById(R.id.icon);
TextView name = (TextView) convertView.findViewById(R.id.text1);

DrawerItem item = (DrawerItem) getItem(position);
icon.setImageResource(item.getIconId());
name.setText(item.getName());

      return convertView;
      }
}
```

Hasta ahora hemos definido los diseños, el modelo y el adaptador necesario para el cajón de navegación. El siguiente paso es comenzar a trabajar sobre el desarrollo de la actividad principal.

3.2.2 Eventos y opciones del Navigation Drawer

La siguiente tarea es rellenar el menú lateral con las opciones que queramos. Para ello primero hay que declarar una instancia del adaptador que contiene los ítems de la lista.

```
//getTitle asocia el titulo de la aplicacion a un parametro.
        mTitle = mDrawerTitle = getTitle();
//Devuelve los recursos asociados y el vector donde se ha definido
//el titulo del menu desplegable
        OpcionesTitles = getResources().
                         getStringArray(R.array.opciones);
//Definicion de atributos creados en el layout
        mDrawerLayout = (DrawerLayout)
                         findViewById(R.id.drawer_layout);
        mDrawerList = (ListView) findViewById(R.id.left_drawer);

//Instancia un archivo layout.xml correspondiente al objeto vista.
        LayoutInflater inflater = getLayoutInflater();
//Inflamos el layout cabecera que contendra el menu lateral.
        ViewGroup header = (ViewGroup)
            inflater.inflate(R.layout.header,mDrawerList,false);
//Colocamos la cabecera en la parte superior del menu lateral.
        mDrawerList.addHeaderView(header, null, false);
```

A continuación, añadimos las opciones y los iconos al cajón de navegación. Previamente debe haberse definido el texto en la carpeta `res/values/string.xml`, para obtenerlo de un array de strings llamado "*OpcionesTitles*", (Ver anexo C) y las imágenes en `res/drawable`. Estos recursos se muestran en la ilustración 26.

```
//Llamamos a un vector que enviaremos a otra  clase donde
// gestionara los iconos y textos del menu lateral
        ArrayList<DrawerItem> items = new ArrayList<DrawerItem>();
        items.add(new
DrawerItem(OpcionesTitles[0],R.drawable.ic_action_settings));
        items.add(new
DrawerItem(OpcionesTitles[1],R.drawable.ic_action_gamepad));
        items.add(new
DrawerItem(OpcionesTitles[2],R.drawable.ic_action_network_wifi));
```

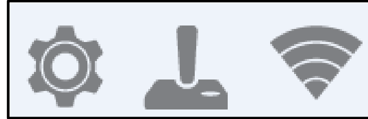


Ilustración 26: Iconos del menú de navegación

```
//Metodo de llamada cuando el boton del menu es pulsado
        mDrawerList.setOnItemClickListener(new
                        DrawerItemClickListener());

//Habilita el icono de la aplicacion en la ActionBar para que se
//comporte como un boton y pueda cambiar la navegacion del cajon
        getActionBar().setDisplayHomeAsUpEnabled(true);
        getActionBar().setHomeButtonEnabled(true);
```

Estas dos últimas líneas de código son una parte fundamental y característica del diseño. Sirven para que el icono de la aplicación muestre una imagen y responda a los “clicks” de apertura y cierre del menú de navegación. Ilustración 27.

Ilustración 27: Botones de acceso al menú

El trozo de código siguiente, gestiona la acción de abrir y cerrar la barra lateral. Usa la interfaz *DrawerListener* para manejar estas acciones y de deslizamiento del Navigation Drawer. Posteriormente se implementa los métodos *onDrawerOpened()* y *onDrawerClosed()*.

`ActionBarDrawerToggle` es un elemento que se implementa en la *Action Bar* para abrir y cerrar un Navigation Drawer con el icono de la aplicación. Normalmente se representa con un *drawable* de tres barras horizontales representadas en la ilustración 27.

El primer parámetro es el contexto donde se ejecuta, el segundo la instancia que se va a realizar, el tercero, la imagen por la que se representará, y por último los dos *strings* de accesibilidad que contiene la información sobre la apertura y cierre del *Drawer.* A través de ellos, cambiaremos los títulos del encabezado dependiendo de si está abierto (nombre de la aplicación), o si está cerrado (nombre de la opción seleccionada).

```
        mDrawerToggle = new ActionBarDrawerToggle(
                this,           /* Actividad donde se desarrolla */
                mDrawerLayout,           /* Objeto DrawerLayout */
                R.drawable.ic_drawer, /*Imagen donde abrir menu */
                R.string.drawer_open,          /* "open drawer" */
                R.string.drawer_close          /* "close drawer" */
                ) {
            public void onDrawerClosed(View view) {
                getActionBar().setTitle(mTitle);
                invalidateOptionsMenu();
// crea la llamada a onPreparateOptions().
            }
            public void onDrawerOpened(View drawerView) {
                getActionBar().setTitle(mDrawerTitle);
                invalidateOptionsMenu();
// crea la llamada a onPreparateOptions().
            }
        };
        mDrawerLayout.setDrawerListener(mDrawerToggle);

        //Al iniciar la aplicacion, mostramos la seleccion 1 del
menu lateral (Ajustes)
        if (savedInstanceState == null) {
            selectItem(1);
        }
```

Una vez declarados los recursos, deberemos gestionar las llamadas al pulsar sobre cada uno de ellos. Cuando un ítem de la lista es pulsado, inmediatamente el contenido debe actualizarse para la selección. Dicha actualización se lleva a cabo a través del método *selectItem()* el cual se invoca dentro de *OnItemClick()*, que requiere el uso de la escucha *OnItemClickListener().* Este método debe remplazar el contenido que se encuentra en el DrawerLayout por un nuevo fragmento adaptado a las condiciones de selección.

```
//Metodo de invocacion de llamada del ListView en el menu lateral
*/
    private class DrawerItemClickListener implements
ListView.OnItemClickListener {
        @Override
        public void onItemClick(AdapterView<?> parent, View view,
int position, long id) {
            selectItem(position);
            //parent. AdapterView donde se ha hecho click.
            //view. Vista seleccionada del vector.
            //position. Posicion de la vista en el adapter.
            //id. identificador del item clicado.
        }
    }
```

El método *selectItem*() recibe como parámetro la posición del ítem seleccionado. En el siguiente apartado, explicaremos los *fragments* y volveremos sobre este método, ya que están implicados.

3.3 Fragmentos

Los fragmentos son entidades con interfaz y lógica de negocio reutilizables a través de las *Activity* y solamente a través de ella deben existir dentro de una actividad, que además se encargará de la gestión del ciclo de vida de dichos fragmentos. Dicho de una manera práctica, con los *fragments*, podemos incluir varias vistas en una sola pantalla y poder interaccionar entre ellas sin tener que cambiar de vistas como si ocurre con las actividades. La ilustración 28 muestra el diseño que tendría la aplicación en un dispositivo móvil y una tableta con los tres fragmentos con los que principalmente trabajaríamos. En el caso de la tableta no nos haría falta el cajón de navegación, ya que tendríamos siempre a la vista todos los fragmentos debido al gran tamaño de la pantalla.

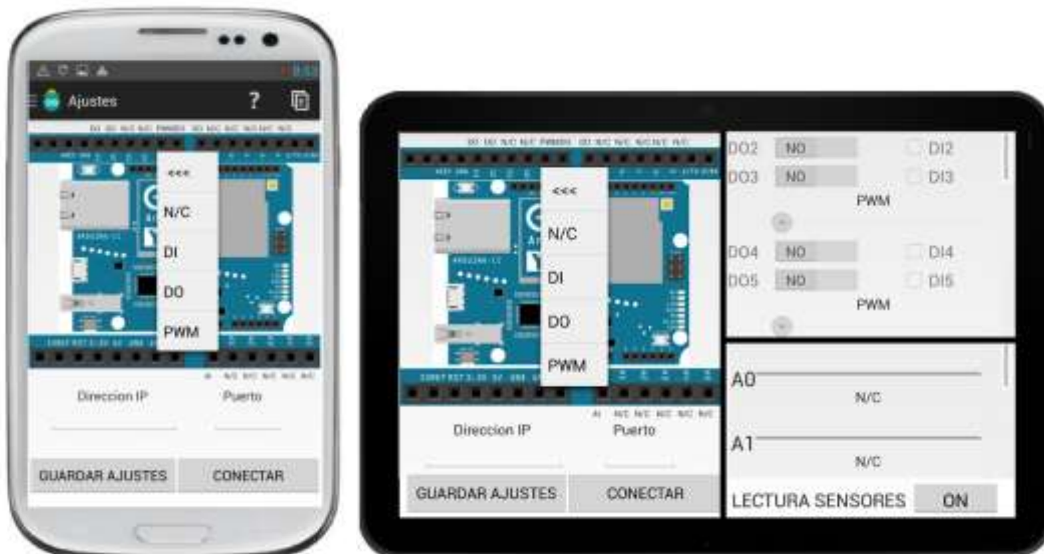


Ilustración 28: Aplicación en smartphone y tablet con *fragments*.

Para crear un *fragment*, es necesario crear una subclase de tipo *Fragment*, que tiene unos métodos de *callback* muy semejantes a las actividades. Por ejemplo *onCreate*(), *OnPause*() …

Para que una actividad pueda alojar un fragmento, vale con extender la clase *Activity* a partir de la versión 3.0 de Android, pero si se quiere en

versiones anteriores, es necesario extender una librería de compatibilidad que ofrece Google y la clase debe extender *FragmentActivity*.

La comunicación entre fragmentos se suele hacer a través de la actividad principal que los acoge, ya que está siempre disponible en todos los fragmentos mediante la llamada `getActivity()`, que devuelve la actividad contenedora en ese momento de ese fragmento y la actividad puede conocer qué fragmentos tiene asociados y buscar uno concreto entre ellos.

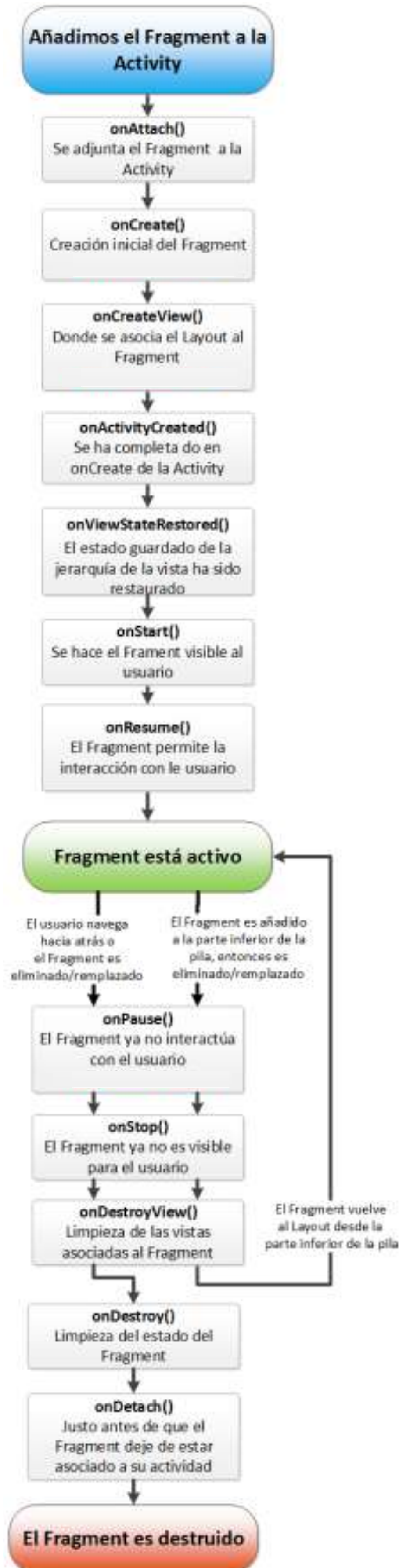
Cada *fragment* ha de implementarse en una clase distinta. Esta clase tiene una estructura similar a la de una actividad, pero con algunas diferencias. La primera es que esta clase tiene que extender `Fragment`. El ciclo de vida dispone de unos cuantos eventos más que la actividad, que le indican cambios en su estado. Este ciclo de vida va asociado al de la actividad que lo contiene. A continuación describiremos el ciclo de vida para conocer su estructura y programar en los bloques adecuados. Posteriormente Introduciremos las partes importantes del código.

3.3.1 Ciclo de vida de un Fragmento

Es importante gestionar el ciclo de vida de un fragmento, ya que depende directamente del estado que pueda adquirir éste al manipular la aplicación. Existen tres estados, uno de ellos ***Reanudado***, donde el fragmento es visible en la actividad que se está ejecutando. El segundo estado, ***En pausa***, indica que otra actividad está en primer plano, pero la actividad en la que este fragmento funciona esta todavía visible. Por último, el estado ***Parado***, indica que el fragmento no es visible. Bien, la actividad se ha parado o bien, el fragmento se ha eliminado de la actividad. Un fragmento parado está todavía vivo, sin embargo no es visible por el usuario y finalizará si la actividad concluye.

Por otra parte, como sucede en las actividades, un fragmento puede retener su estado mediante el método *Bundle* en caso de que el proceso de la actividad termine y se necesite posteriormente restaurar el estado del fragmento cuando se vuelva a crear la actividad. Se puede guardar el estado durante el callback *onSaveInstanceState*() y restaurarlo durante el *onCreate*(), el *onCreateView*() o el *onActivityCreated*().

En resumen, el ciclo de vida de un *fragment* está ligado al de la *activity*. Cuando ésta última se pause o se destruya, se pausarán o se destruirán todos los fragmentos que incluya. Si la *activity* se encuentra en ejecución, puede manipular independientemente cada *fragment*.



onAttach: Es invocado cuando el fragmento ha sido asociado a la actividad

onActivityCreated: Se ejecuta cuando la actividad ya ha terminado la ejecución en su método *OnCreate*().

onCreate: Este método es llamado cuando el fragmento se está creando. En él podemos inicializar todos los componentes que deseemos guardar si el fragmento se pausa o se detiene.

onCreateView: Se llama cuando el fragmento será dibujado por primera vez en la interfaz de usuario. En este método creamos el View que representa al fragmento para retornarlo hacia la actividad.

onStart: Se llama cuando el fragmento esta visible ante el usuario. Depende de la actividad.

onResume: Es ejecutado cuando el fragmento esta activo e interactuando con el usuario. Depende si la actividad principal está en estado *onResume*.

onStop: Se llama cuando un fragmento ya no es visible para el usuario debido a que la actividad principal se ha detenido o se está gestionando otros fragmentos.

onPause: se ejecuta cuando se detecta que el usuario dirigió el foco por fuera del fragmento

onDestroyView: Este método es llamado cuando la jerarquía de Views a la cual ha sido asociado el fragmento ha sido destruida.

onDetach: Se llama cuando el fragmento ya no está asociado a la actividad.

3.3.2 Implementación de un Fragmento

Un fragmento, como ya definimos, es un trozo de la interfaz de usuario que se coloca en la actividad principal. La interacción entre los fragmentos se realiza a través de *FragmentManager* que se puede obtener por *Fragment.getFragmentManager ().* Para gestionar los fragmentos disponemos de diferentes métodos como *findFragmentById* o *findFragmentByTag.* A continuación mostramos las declaraciones implementadas en el código.

En primer lugar, debemos crear la gestión y manejo de los fragmentos desde la actividad principal. Con el menú de navegación, accedemos a los fragmentos mediante el método SelectItem().

Este método trata la posición que el usuario ha seleccionado en la ListView del menú de navegación y devuelve el número de la posición que ocupa.

```
//Funcion que gestiona la posicion del item seleccionado
    private void selectItem(int position) {
        //Declaracion del parametro fragment como null.
        Fragment fragment = null;
```

La posición cero está declarada como la cabecera, por lo tanto no trabajaremos con ella y no tendrá eventos ni escuchadores. Por el contrario, respecto a las posiciones 1, 2 y 3, corresponden a los *fragments* de ajustes, controles y sensores respectivamente. Así cuando pulsemos por ejemplo en ajustes, el método nos devolverá la posición uno y con la función *switch* accederemos a la clase *AjustesFragment().*

```
        //Dependiendo de la posicion seleccionada accederemos a un
fragment determinado.
        switch (position) {
        case 1:
            fragment = new AjustesFragment();
            break;
        case 2:
            fragment = new ControlesFragment();
            break;
        case 3:
            fragment = new SensoresFragment();
            break;

        default:
            break;
        }
```

3.3.3 Gestión de los Fragmentos

Para realizar la transición entre los fragmentos, utilizamos `FragmentManager`. Cuando comprobamos que el fragmento no está vacío, se realiza la gestión y transición entre los demás para visualizarlo por pantalla. Existen tres métodos *add*, *replace* y *remove*. El primero añade el *fragment* a la parte superior de la vista, superponiendo ambas. El segundo remplaza la vista, eliminando la anterior. El tercero, elimina el *fragment* que se vaya a instanciar. Seguidamente para aplicar la transición a la actividad debemos llamar a *commit*().

```
        //Si el fragment no esta vacio, los gestionamos a traves
de una interfaz.
        if (fragment != null) {
            FragmentManager             fragmentManager          =
getFragmentManager();

fragmentManager.beginTransaction().replace(R.id.content_frame,
fragment).commit();
```

Como describimos en el diseño del `activity_main` o actividad principal del layout, el identificador `content_frame`, se define para el *FrameLayout* que mostrará cada fragmento que reciba la llamada. Para nuestra aplicación, remplazamos con el método *replace* y lo mandamos al identificador del *FrameLayout* de la actividad para que muestre el nuevo layout en pantalla completa.

Una vez gestionada la transición de los fragmentos es hora de declarar cada clase y estructurarla de acuerdo al ciclo de vida que resumimos anteriormente. Para instanciar un *fragment* es necesario hacer una llamada a *`new`* seguida de la nueva clase desde la actividad principal para acceder a esta nueva.

```
 fragment = new AjustesFragmet();
fragment = new ControlesFragmet();
fragment = new SensoressFragmet();
```

3.3.4 Creación de un Fragmento

Ahora sí, podemos empezar a trabajar en nuestra nueva clase AjustesFragment.java. En todas ellas la estructura será semejante. Se indicarán las diferencias que en cada una pueda existir. En la primera clase

debemos extenderla al método *Fragment*. Posteriormente iniciamos las declaraciones de los objetos y vistas que participaran en ella e iniciamos un constructor vacio público para no ocultar la visibilidad del fragmento y no provocar errores en la aplicación.

```
public class AjustesFragment extends Fragment {

      /*****************************************************************/
      /*          Definicion de los atributos de la clase         */
      /*****************************************************************/

      public AjustesFragment() {
    }
```

Ahora nos vamos a saltar la estructura que describíamos en el ciclo de vida para explicar de manera continua la creación de un fragmento. En apartados siguientes se muestran otros métodos que se asocian para la creación, por ejemplo, de la comunicación entre fragmentos, pero que trataremos más adelante.

El siguiente método onCreateView() lo llamamos para dibujar por primera vez la interfaz. Para ello, el fragmento debe devolver un View desde este método.

```
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
                    container,  Bundle  savedInstanceState)  {

      Log.v(TAG, "In frag's on create view");
      final View v = inflater.inflate(R.layout.fragment_ajustes,
                    container, false);

/*****************************************************************/
/*        Definicion e inicializacion de los objetos.       */
/*****************************************************************/
      return v;
    }
```

Éste es el único método que tratará esta clase. En él, se describirá el código que crea la interfaz y define cada objeto. Después de la inicialización de vistas y objetos comenzamos a declarar cada uno de ellos.

El concepto *ArrayAdapter* ya fue explicado anteriormente, por lo que nos saltaremos este paso. Los parámetros que definen son el contexto donde se incluye, la vista con la que se mostrará el *array* (un *spinner* como declaramos en el layout) y por último el *string* con los datos que contendrá.

```
lblMensaje2 = (TextView)v.findViewById(R.id.LblMensaje2);
            spinner2 = (Spinner)v.findViewById(R.id.spinner2);

//vector de datos que contiene los parametros necesarios para implementarlo en el Spinnner
        final String[] datos2 =
                new String[]{" <<<","N/C","DI","DO"};

//Asociar el vector con la actividad actual y el objeto declarado en el layout
//para mostar los datos del vector creado en el spinner concreto.
        ArrayAdapter<String> adaptador2 =
            new ArrayAdapter<String>(getActivity(),
                      android.R.layout.simple_spinner_item,datos2);
```

El siguiente paso es asociar cada dato o elemento del spinner a una acción concreta. Este paso lo realizan los escuchadores, quienes detectan el elemento que ha sido pulsado y lo envían al método *onItemSelected*() que está contenido en el escuchador *setOnItemSelectedListener*().

```
//Devolucion de la llamada del spinner una vez que ha sido pulsado.
        spinner2.setOnItemSelectedListener(
//El spinner ha sido seleccionado y muestra la interfaz que definimos antes.
                new AdapterView.OnItemSelectedListener() {
//Metodo invocado que recibe la posicion inicial, la posicion actual o del dato pulsado, el layout creado en el xml y su identificador(id)
public void onItemSelected(AdapterView<?> parent,
      android.view.View v, int position,long id) {

//Con la funcion getItemAtPosition conseguimos un entero int, que nos da la posicion del dato en la lista SetText nos permite pasar a String la posicion seleccionada y asociarla a un textview para que se visualice en el layout.
      lblMensaje2.setText(""+parent.getItemAtPosition(position));

public void onNothingSelected(AdapterView<?> parent) {
      lblMensaje2.setText("");
                   }
                });
```

Repetimos los pasos para completar las declaraciones con los restantes pines que necesitemos configurar. El fragmento básico estaría completo a falta de la comunicación entre ellos y la actividad.

El siguiente *fragment* se compone de la misma estructura, extendiendo la clase Fragment e implementando los métodos de *SeekBar.ChangeListener* y *View.onClickListener* para implementar y manejar el uso de los botones *switch* y la barra de progreso deslizante, *seekbar*.

```
public class ControlesFragment extends Fragment implements
      SeekBar.OnSeekBarChangeListener, View.OnClickListener{
```

En el caso de la clase `ControlesFragment.java`, tendremos el método *onCreateView*(), en el cual se declararán las vistas y objetos que formen el layout y la clase. Así de esta forma, evitamos cada vez que abramos esta clase, que nos cree desde cero la interfaz y podamos recuperar mediante el método *onStart*() el estado anterior.

```
@Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState) {

            View v = inflater.inflate(R.layout.fragment_controles,
                        container, false);

      /**********************************************************/
            /* Declaracion de las vistas que forman el layout */
      /**********************************************************/
      return v;
}
```

Una vez, implementada la interfaz, añadimos el método onStart() que es llamado cuando el fragmento ha sido creado y está visible ante el usuario. Aquí definimos los *switches*, *seekbar* y botones que contenga el *fragment.*

```
      @Override
    public void onStart() {
        super.onStart();
}
```

La parte del código que contiene son tareas asíncronas que describiremos en detalle en un apartado propio debido a su complejidad.

En cuanto a `SensoresFragment.java` dispone la misma estructura que el último *fragment* explicado, con los métodos *onCreateView*() y *onStart*(). En este caso, no haría falta implementar ningún método en la clase *Fragment,* ya que solo trataremos con progressbar y no manejaríamos su estado.

3.3.5 Comunicación con Fragmentos

Para reutilizar los componentes *Fragment*, debemos desarrollar cada uno de ellos como componentes modulares que definan su propia interfaz y comportamiento. Una vez definidos podemos asociarlos con una *Activity* y conectarlos con la lógica de la aplicación para crear una interfaz de usuario compuesta.

La comunicación con fragmentos se realiza cuando queremos cambiar el contenido de uno de ellos basándonos en un evento del usuario. Ésta comunicación se lleva a cabo a través de la actividad asociada por lo que dos fragmentos no pueden comunicarse directamente.

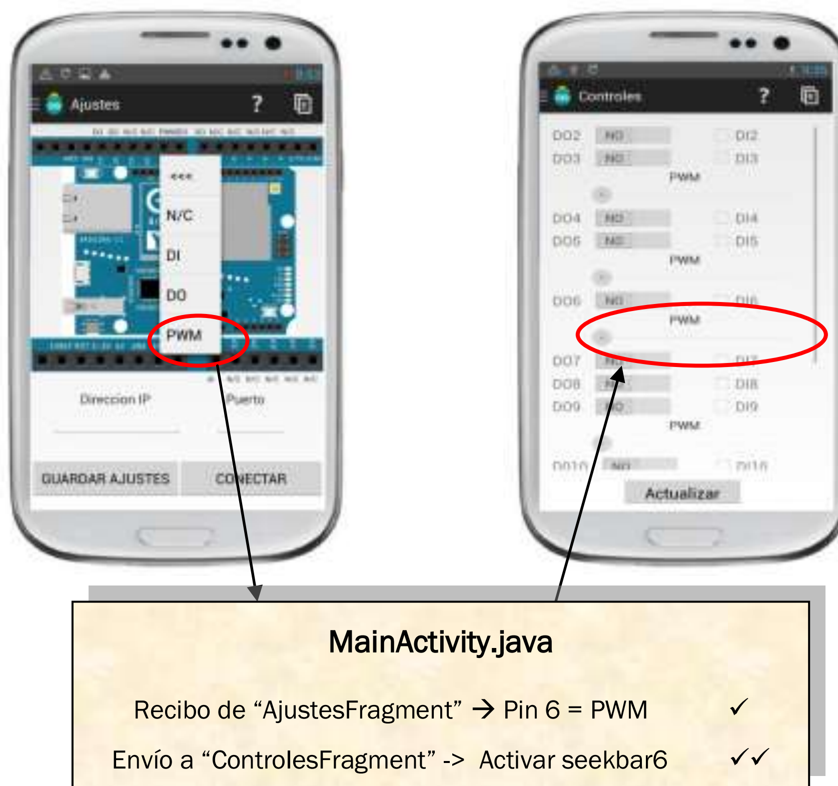


Ilustración 29: Paso de parámetros a través de la actividad de acogida.

El parámetro del *fragment* "Ajustes", se envía a través de una interfaz a la actividad principal de acogida que posteriormente manda ese parámetro al nuevo *fragment* "Controles", donde actualiza y aplica el nuevo valor. En este caso, indicamos que el pin 6 se configure como salida analógica y por tanto, activa su manejo en el segundo layout.

Pero para conseguir lo anterior, es necesario realizar unos pasos previos. Primero debemos definir una interfaz en la clase *Fragment* e implementarla en la actividad. El fragmento captura la implementación de la interfaz durante su método de ciclo de vida *onAttach*(), y puede luego llamar a los métodos de la interfaz para comunicarse con la actividad.

Clase Fragment

Lo primero que necesitamos es declarar una interfaz, de esta manera cada vez que el fragmento reciba un evento también lo recibirá la actividad, que se encargar de recibir y enviar los datos a otros fragmentos. A esta interfaz la hemos llamado *OnFragmentClickListener* y contiene un método llamado *onFragmentClick*() con unos parámetros que podemos ver a través del anexo C.

```
/*************************************************************/
/* Interfaz creada para el paso de variables entre fragmentos*/
/*********************************************************** */

    public  interface OnFragmentClickListener{
        public void OnFragmentClick(/*parametros*/);
    }
```

Ahora el fragmento puede enviar mensajes a la actividad llamando al método `onFragmentClick()` usando la instancia `mListener` de la interfaz `onFragmentClickListener()`.

Para comprobar que la actividad de acogida implementa esta interfaz, sobrescribimos el método *onAttach()*, que será llamado cada vez que la actividad cree una instancia del fragmento.

```
/*************************************************************/
/* Metodo de llamada para adjuntar el fragment al Activity. */
/*************************************************************/
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        try{
            mListener = (OnFragmentClickListener) activity;
        }catch(ClassCastException e){
            throw new ClassCastException(activity.toString() +
                        " must implement listeners!");
        }
    }
public void onClick(View v){
//Hacemos una llamada al método para enviar las variables
      mListener. onFragmentClick(/*parametros*/);}
```

Esta última línea de código, se encarga de enviar el evento y los datos a la interfaz. Para ello es preciso una instancia `mListener` que se implementa en el método *onClick* para atender el evento cuando el escuchador es llamado, es decir, cuando el usuario pulse el botón "Guardar ajustes", se invocará el método y actuará la instancia `mListener` enviando los parámetros a la actividad.

Clase Activity

Para recibir la retrollamada de eventos de un fragmento, la actividad que lo contiene debe implementar la interfaz definida anteriormente en el fragmento. Se define junto al nombre de la clase.

```
public class MainActivity extends FragmentActivity implements
      AjustesFragment.OnFragmentClickListener {
            /*resto del codigo*/
}
//La Interfaz creada en el Fragment Ajustese necesita pasar por la
actividad principal para enviarla a otro fragmento.
//Esta funcion recibe los parametros a pasar al fragment.
      @Override
      public void OnFragmentClick((/*parametros*/) {/*resto del
                                              codigo*/}
```

En el método, guardamos los datos que recibimos de cada evento del *fragment* en un *Bundle* y se lo pasamos a la nueva clase con `FragmentManager`, llamando a `ControlesFragment()`

```
      ControlesFragment controlfrag = new ControlesFragment();
//Inicializa los argumentos a pasar al fragment
            Bundle args = new Bundle();

//Enviamos las variables al fragment declarado anteriormente y le
asignamos un nombre clave.
//Con setArguments enviamos los argumentos que necesitemos
          args.putString(controlfrag.CONFIG_PIN, p2);
          controlfrag.setArguments(args);

// Declarar primero setArguments antes que iniciar el fragment
//ya que si no los argumentos a traves de Bundle no se pasarán.
//Ahora remplazamos el fragment actual por el fragment destino que
queremos mostrar.
          getFragmentManager().beginTransaction()
              .replace(R.id.content_frame, controlfrag).commit();

      }
```

Ahora sí, podemos comunicarnos entre fragmentos a través de la actividad de acogida. Los parámetros que pasaremos en nuestro caso, serán los *string* que nos muestra el *TextView* y que tendremos la posibilidad de configurarlos como “N/C (No Conectado)”, “DO (Digital Output)”, “DI (Digital Input)”y “PWM (Pulse With Modulataion o salida analógica)”.

3.4 Hilos de ejecución. Tareas asíncronas

Cada vez que lanzamos una nueva aplicación en Android, el sistema crea un nuevo proceso Linux para ella y la ejecuta en su propia máquina virtual Dalvik. Todos los componentes se ejecutan en el mismo proceso.

Los SO modernos incorporan el concepto de hilo de ejecución (*thread*). En un sistema multihilo, un proceso va a poder realizar varias tareas a la vez, cada una en un hilo diferente. Los diferentes hilos de un proceso lo comparten todo: variables, código, permisos, ficheros abiertos, etc. Si trabajamos con varios hilos, éstos pueden acceder a las variables de forma simultánea.

Cuando se lanza una nueva aplicación, el sistema crea un nuevo hilo de ejecución (*thread*) para esta aplicación, conocido como hilo principal. Este hilo es muy importante, dado que se encarga de atender a los eventos de los distintos componentes generados desde la interfaz de usuario.

Si realizamos un trabajo intenso debemos crear nuestros propios hilos ya que si no estaremos bloqueando el hilo principal. Con ello surge la clase ***AsyncTask***. Una tarea asíncrona permite realizar un cálculo o proceso que se ejecuta en un hilo secundario, permite también realizar operaciones de bloqueo en un hilo secundario y publicar los resultados en la interfaz grafica.

Esta tarea asíncrona debe cumplir unos requisitos:

- Hilo secundario
  - Método ***doInBackground***: Será el que se ejecute en dicho hilo.
- Hilo principal o de la interfaz grafica. Los siguiente métodos se ejecutaran en dicho hilo:
  - Método ***onPreExecute***: Se ejecuta antes de que tenga lugar el método *doInBackground*.
  - Método ***onProgressUpdate***: Se ejecuta cada vez que desde el método *doInBackground* queramos publicar el progreso con el método ***publishProgress***
  - Método ***onPostExecute***: Se ejecuta al finalizar el método doInBackground

- La comunicación durante doInBackground:
  El método ***publishProgress*** se ejecutará desde el hilo secundario, para notificar al hilo principal que debe llevar a cabo la acción determinada ***onProgressUpdate.***
- La comunicación a lo largo del ciclo de vida se realiza a través de parámetros.

Para iniciar un *AsynTask* basta con hacer uso del constructor y llamar al final al método *execute.* Éste es un método asíncrono, lo que significa que, tras llamarlo, se pondrá en marcha la tarea en otro hilo, pero en paralelo se continuarán ejecutando las instrucciones que hayamos escrito después de *execute*.

```
//Creacion de una tarea asincrona. Necesaria para evitar que una
tarea bloquee
            //el hilo principal de la interfaz de ususario.
            new AsyncTask<Void, Void, Void>() {

//Llamada despues del OnPreExecute(En este caso no era necesario
implementarlo)
//Es aqui donde se realiza la tarea principal
                  @Override
                  protected Void doInBackground(Void... params) {
HttpResponse respone = httpclient.execute (new HttpGet (url));
//resto del codigo
                  }

//Una vez terminado el metodo anterior, llamamos a onPostExecute
                  @Override
                  protected void onPostExecute(Void aVoid) {
//Realizamos una función
                  }
              }.execute();
          }
```

3.5 Protocolo HTTP

En Internet la aplicación más destacada es la World Wide Web (www) . La web nos ofrece un servicio de acceso a información distribuida en miles de servidores de todo Internet. Para la comunicación entre los clientes y servidores de esta aplicación, se emplea el protocolo *HTTP* (Hyper Transfer Protocol).

HTTP es un sencillo protocolo cliente servidor que articula los intercambios de información entre los navegadores web y los servidores web. En la web, los servidores han de escuchar en el puerto 80, esperando la conexión de algún cliente web.

En el código Java, debemos crear el objeto *HttpClient*. Después, para ejecutar la solicitud, devuelve un objeto *HttpResponse*, cuya información se extrae y analiza. Seguidamente debemos tomar las excepciones *IOException* y *ClientProtocolException*, para el caso de conexión fallida y por error de protocolo, respectivamente.

```
//HttpClient crea una interfaz para un cliente HTTP.
//El cliente encapasula los objetos necesarios para ejecutar las
peticiones HTTP durante la autenticacion, gestion de la conexion y
otras caracteristicas
      HttpClient httpclient = new DefaultHttpClient();

//try/catch es un bloque que gestiona los fallos que se pueden
ocasionar dentro del bloque try colocamos las funciones que
podrian provocar fallo, si esto ocurre, saltamos al bloque catch
que gestionara el error.
                      try {

//Recibimos la respuesta HTTP y ejecutamos nuestra peticion.
//Enviamos la URL que interpretará Arduino
HttpResponse respons = httpclient.execute(new
                                     HttpGet(web_service13o));
HttpResponse response = httpclient.execute(new
                                     HttpGet(web_serviced13));
//Esta clase implementa un flujo de salida en el que los datos se
//escriben en una matriz de bits (32bits por defecto). El buffer
crece automáticamente como se escriben datos en ella.

ByteArrayOutputStream out = new ByteArrayOutputStream();
//Escribe los datos obtenidos en el flujo y escribe en la cadena
enviada.
response.getEntity().writeTo(out);
//cierra el flujo de datos
out.close();

                      } catch (Exception e) {
                          e.printStackTrace();
                      }
                      return null;
                  }
```

Debido al tiempo que las conexiones a Internet puedan tardar en devolvernos los resultados, tenemos que ejecutar en un hilo secundario la operación para no bloquear el hilo principal `httpclient.execute`. Con la clase *AsyncTask* que explicamos en el apartado anterior, solicitamos y/o recibimos (en caso de enviar a Arduino o recibir de éste un comando) una petición mediante una URL definida al comienzo del programa. Esa URL será por defecto *http://192.168.240.1/arduino/* seguido de la instrucción que deseemos, bien sea analógica o digital, seguida del pin y en caso de escritura, acompañada de un valor entre 0-255 ó 0-1, respectivamente.

Para realizar la escritura en Arduino utilizamos el objeto *HttpGet* y con una variable auxiliar, añadimos el valor de escritura a la URL formando la cadena completa.

```
HttpGet httpget = new HttpGet(URL);
aux= URL + "/" + aux;
HttpGet httpget = new HttpGet(aux);
```

Con esta última declaración recuperamos la cadena completa y que Arduino entenderá y procesará mediante su código. El protocolo *HTTP* se ha cumplido y podemos recibir y enviar peticiones sin olvidarnos de los permisos que se deben incluir en el fichero *AndroidManifest.xml* que veremos más adelante.

3.6 Almacenamiento de datos

Leer y escribir archivos en Android es muy útil si queremos interaccionar con nuestra aplicación y modificar archivos de texto. En este caso, almacenamos ficheros *.txt* en la sdcard o tarjeta SD externa.

Los archivos guardados en el almacenamiento externo son legibles para todas las aplicaciones y pueden ser modificados por el usuario en cualquier momento y a través del almacenamiento masivo USB, que permite la transferencia de archivos al ordenador.

Es necesario establecer unos permisos en el fichero *AndroidManifest.xml* tanto de lectura como de escritura si ese fuera el caso, para almacenar datos en el sistema.

En cuanto al código Java, hemos creado la clase `FileOperations.java` que realiza las operaciones de lectura y escritura. Para la operación de escritura, si el archivo nuevo no existe, creará un nuevo. El contenido se escribe utilizando *BufferedWritter*. Para leer el contenido del fichero utilizamos la operación *BufferedReader*.

```
public class FileOperations {

    public FileOperations() {
    }
    //Escritura en el fichero en la ruta determinada.
    public Boolean write(String fname,String fpin){
      try {
          //Si el fichero no existe lo crea
```

```
            if (!file.exists()) {
                file.createNewFile();
            }

//FileWriter escribe un archivo en una parte especifica del sistema
//getAbsoluteFile devuelve un archivo nuevo usando la ruta absoluta.
FileWriter fw = new FileWriter(file.getAbsoluteFile());
BufferedWriter bw = new BufferedWriter(fw);
            bw.write(fpin);
            bw.close();
            Log.d("Suceess","Sucess");
            return true;
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }

      }
//Lectura del fichero con el nombre guardado en la variable "fname".
      public String read(String fname){
        try {
            //Inicializa el flujo de datos
            StringBuffer output = new StringBuffer();
            String fpath = "/sdcard/Arduino_Yun/"+fname+".txt";
            br = new BufferedReader(new FileReader(fpath));
            String line = "";
            //Lee hasta que finalice la linea del fichero.
            while ((line = br.readLine()) != null) {
                output.append(line);
            }
            br.close();
            response = output.toString();

        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
        return (response);
      }
}
```

Cuando se pulsa el botón de guardar, el escuchador escucha la llamada y realiza la acción de invocar la clase `FileOperations()`. El archivo se crea en la tarjeta SD y se guardan los datos declarados en el *string*.

```
save.setOnClickListener(new OnClickListener(){
```

```
            @Override
            public void onClick(View v) {
//Declaracion local de las variables a guardar en el telefono.
//getText nos devuelve el valor que contiene el textview
//toString devuelve un String con los mismos caracteres que visualizabamos.
                  String p2 = lblMensaje2.getText().toString();
                  ............
//Realizamos una llamada a una nueva clase.
                FileOperations fop = new FileOperations();
//Enviamos los parametros que definiamos anteriormente como variables
                  //locales para guardarlos en memoria.
                fop.write(pin2,p2);
```

Para realizar la lectura, al inicio del *fragment*, introducimos unas declaraciones que invocan a esta clase, en concreto al método de lectura y actualiza los *TextView* y demás variables que estén asociadas a la configuración de los pines, como la activación de los botones.

```
//Crea una nueva clase para realizar la invoocacion posterior.
        FileOperations fop = new FileOperations();
        //Envio de parametros a la nueva clase
        String carga2 = fop.read(pin2);

//Condicion necesaria para comprobar si existen los ficheron que queremos leer.
        if(carga2 != null){
//Si el fichero existe, leemos de la memoria interna y le asociamos ese valor al texview que se creo a traves del layout
            txtprueba1.setText(carga2);
            //Asignamos ese valor a una variable
            p2=txtprueba1.getText().toString();

//CompareTo compara los String recibidos, devolviendo el valor cero si son iguales, 1 s es menor o 1 si es mayor.
            if(p2.compareTo(NC) != 0 && p2.compareTo(DI) != 0){
                  btn2o.setEnabled(true);
            }
        }
```

Gracias a esto, ahora en nuestra tarjeta de memoria tendremos varios ficheros *.txt* correspondientes a la configuración de los pines. Cada vez que salgamos de la aplicación, no perderemos los datos y al entrar de nuevo en ella, la configuración seguirá mostrándose en el *fragment*. Por otra parte, también podemos cambiar los parámetros de los pines sin acceder a la aplicación, desde el sistema de almacenamiento de nuestro dispositivo y con un editor de texto.

3.7 AndroidManifest.xml

Este archivo se encuentra en la carpeta `src/main`, dónde se declara cómo es internamente la aplicación, qué actividades la componen, qué servicios existen, etc. El archivo es creado automáticamente cuando se crea un nuevo proyecto y se incluyen en él las opciones básicas para que una aplicación de una sola actividad y sin permisos especiales funcione sin problemas. Cuando generamos aplicaciones Android, tendremos que ir añadiendo opciones a este fichero con tal de poder navegar entre actividades, lanzar servicios o conectaremos a internet por ejemplo.

A través de los filtros de intenciones, este fichero, nos determinará cómo interactuará la aplicación. Los elementos que componen son:

- ***uses-permission***: Indican los permisos que la aplicación necesita para funcionar de modo correcto.
- ***permission***: Indican permisos que las actividades o servicios pueden requerir a otras aplicaciones para poder acceder a elementos de la aplicación, como por ejemplo a los datos.
- ***instrumentation***: Sirve para indicar las clases que se deben invocar para cuestiones de monitorización y registro de *Logs*.
- ***uses-sdk***: Versión de Android para la cual ha sido diseñada la aplicación. Tiene un atributo llamado minSdkVersioon para informar del nivel mínimo de API de Android y no provocar que una aplicación no funcione en un dispositivo por el hecho de no tenerlo actualizado.
- ***support-screens***: Se especifica qué tamaño de pantalla soporta la aplicación y cuáles no.
- ***aplication***: Aquí se encuentran definidas todas las *Activity*, *Services*, etc.

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.navigationdrawerexample"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-sdk
android:minSdkVersion="14"
android:targetSdkVersion="21" />

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

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

    <application
        android:allowBackup="true"
        android:label="@string/app_name"
        android:icon="@drawable/ic_launcher_ofi"

android:theme="@android:style/Theme.Holo.Light.DarkActionBar">

        <activity
            android:name=".FragmentSplashActivity"
            android:screenOrientation="portrait"
            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=".MainActivity"
            android:screenOrientation="portrait">
        </activity>

        <activity
            android:name=".Tutorial"
            android:screenOrientation="portrait">
        </activity>

    </application>

</manifest>
```

`android:versionName="1.0` Indica la versión más reciente que existe. Cuando se modifique la aplicación para realizar mejoras, esta declaración, por ejemplo, podría pasar a ser la versión 2.0 u otro cualquier número mayor que 1.0. Así el usuario podrá saber que se ha actualizado la aplicación.

`android:minSdkVersion="14" android:targetSdkVersion="21"` son los niveles mínimo y máximo de la API de Android para el uso de la aplicación.

`android:allowBackup` Si el valor es "*true*" indica que la aplicación debe guardar una copia de seguridad.

`android:label` y `android:icon` son el texto de la aplicación y el icono que

se mostrarán en el menú principal de Android. Ambos recursos deben estar definidos en el fichero `string.xml`

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

Estos son los permisos necesarios para poder establecer la conexión y comunicación a través de Internet y la lectura y escritura en la memoria externa del dispositivo.

Otra de las partes más relevantes de este fichero son los filtros de intenciones que declaremos dentro de `<intent-filter></intent-filter>` y de las actividades `<activity> </activity>`

Por cada actividad que tengamos en nuestra aplicación, incluiremos ésta entre las etiquetas `<activity> </activity>`

Con las siguientes declaraciones, podemos mostrar o lanzar (*LAUNCHER*) la actividad o fragmento que queramos al inicio de la aplicación, como primera vista (*MAIN*). En nuestro caso, iniciamos con el fragment temporal de la firma del autor.

```
<intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

3.8 Código Arduino. Librería Bridge

En este nuevo apartado, tratamos el código implementado en Arduino que dedicará la comunicación entre los procesadores ATmega, que gestiona la tarjeta y Linino, que establece la conexión WiFi. También describiremos las funciones internas para desarrollar el manejo de la placa.

La librería Bridge permite la comunicación en ambas direcciones actuando como una interfaz de línea de comandos de Linux.

El siguiente código muestra cómo es posible hacer peticiones *HTTP* al Yun para leer y escribir información en los pines de la placa. En primer lugar, es necesario incluir las librerías *Bridge*, *YunServer* y *YunClient* e instanciar un servidor que permita al Yun escuchar a los clientes conectados.

```
#include <Bridge.h>
#include <YunServer.h>
#include <YunClient.h>

YunServer server;
```

En el bloque *setup*(), iniciamos la comunicación en serie a 9600 *baudios*. Encendemos y apagamos el pin 13 para indicar que comienza la instrucción *Bridge*

```
void setup() {
  // Bridge inicio
  pinMode(13, OUTPUT);
  digitalWrite(13, HIGH);
  digitalWrite(13, LOW);
  Bridge.begin();
```

A continuación, indicamos a la instancia *YunServer* que debe escuchar solo las conexiones entrantes procedentes de *LocalHost*. Las conexiones realizadas a Linux pasarán al procesador 32U4 para analizar y controlar los pines. Iniciamos el servidor con la instrucción *server.begin*().

```
  // Escucha la conexión solo del LocalHost
  // nunca de una red externa
  server.listenOnLocalhost();
  server.begin();
}
```

En el bloque *loop*() creamos la instancia del *YunClient* para la gestión de la conexión. Si el cliente se conecta, procesa las solicitudes y cierra la conexión cuando finalice. Al final del bloque colocamos un *delay* (retraso) de 50 milisegundos para evitar que el procesador trabaje demasiado. La contrapartida, es el pequeño retraso que podemos recibir al enviar las peticiones *HTTP* no siendo éstas instantáneas.

```
void loop() {
  // Cliente conectado actúa como servidor
  YunClient client = server.accept();
  // nuevo cliente?
  if (client) {
    // Procesa la peticion
    process(client);
    // Cierra la conexión y libera los recursos
    client.stop();
  }
  delay(50); // Retardo de 50ms
}
```

A continuación, se crean diferentes funciones para manejar las conexiones. Una de ellas acepta el YunClient como argumento. Lee el comando mediante la creación de una cadena y analiza los comandos “digital”, “analog”, “mode” y pasa la información a la función que lleva ese nombre.

```
void process(YunClient client) {
  // lee el comando
  String command = client.readStringUntil('/');
  // comando "digital"?
  if (command == "digital") {
    digitalCommand(client);
  }
  // comando "analog"?
  if (command == "analog") {
    analogCommand(client);
  }
  // comando "mode" ?
  if (command == "mode") {
    modeCommand(client);
  }
}
```

La siguiente función, *digital* se crea para manejar los pines digitales de Arduino. Recibe el cliente como argumento y crea las variables locales “*pin”* y “*valor*”. Si el carácter después del pin es “/” significa que la dirección URL va a tener un valor de 0 ó 1. Si no existe nada después de “/” lee el valor del pin.

```
void digitalCommand(YunClient client) {
  int pin, value;

  // lee el numero del pin
  pin = client.parseInt();

  if (client.read() == '/') {
    value = client.parseInt();
    digitalWrite(pin, value);
  }
  else {
    value = digitalRead(pin);
  }
}
```

El mismo caso ocurre con la función *analog,* que gestiona los pines analógicos. Incluye las mismas variables locales y la única diferencia es que al escribir en los pines, la URL tendrá un valor, después del último carácter “/” entre 0 – 255. La lectura alcanzara valores entre 0 – 1023.

http://192.168.240.1/arduino/analog/pin -> Lectura

http*://192.168.240.1/arduino/analog/pin/valor* -> Escritura

```
void analogCommand(YunClient client) {
  int pin, value;

  // Lee el numero del pin
  pin = client.parseInt();

  if (client.read() == '/') {
    // Lee el valor y ejecuta el comando
    value = client.parseInt();
    analogWrite(pin, value);
  }
  else {
    // Lee el pin analogico
    value = analogRead(pin);
  }
}
```

La función *mode* trata de configurar el pin digital o analógico como entrada o salida. Utilizamos esta función si vamos a cambiar la configuración de los pines con frecuencia. Esta vez, deberemos incluir los términos ***input*** u ***output*** después del modo de configuración ("mode/"), siendo *mode* (*analog* o *digital).*

```
void modeCommand(YunClient client) {
  int pin;

  // Lee el pin
  pin = client.parseInt();

    if (client.read() != '/') {
    client.println(F("error"));
    return;
  }

  String mode = client.readStringUntil('\r');

  if (mode == "input") {
    pinMode(pin, INPUT);
    return;
  }

  if (mode == "output") {
    pinMode(pin, OUTPUT);
    return;
  }

  client.print(F("error: invalid mode "));
  client.print(mode);
}
```

3.9 Funciones de Arduino

En este apartado vamos a explicar los recursos necesarios de programación de Arduino para que realice operaciones como la escritura, lectura y configuración de pines, entre otras muchas.

pinMode()

Configura un pin como entrada o salida. Para utilizar esta función, pasamos el número del pin que vamos a configurar y la constante *INPUT* (entrada) u *OUTPUT* (salida). Cuando el pin esté configurado como entrada, podemos detectar el estado de sensores de temperatura, *LDR*, pulsadores... Si es configurado como salida, podemos manejar actuadores, como un *LED*, un piezoeléctrico, un servo, etc.

digitalWrite()

Escribe un valor digital en un pin. Por ejemplo, `digitalWrite(pin, HIGH);` Debemos configurar el numero del pin a la variable *`pin`* y el valor *HIGH* para obtener en la salida +5Voltios. Escribiendo *LOW* conecta el pin a tierra, o a 0 voltios. Esta función se aplica cuando el pin se ha definido como salida digital

digitalRead()

Lee el estado de un pin digital. La instrucción `digitalRead(pin);` nos permite leer el valor que actualmente tiene la variable *pin.* Estos valores serán 1 ó 0. En este caso el pin debe ser digital y el modo puede ser indistintamente tanto entrada como salida. Por defecto Arduino establece los pines como entrada, por ello en ese caso no es necesario declararlos, aunque siempre es conveniente hacerlo para tener un código claro.

analogWrite()

Escribe un valor analógico en un pin específico. En esta función los valores que recibe están comprendidos entre 0 – 255. Un ejemplo, `analogWrite(pin, valor);` El máximo valor que alcanzaremos será con 255, por el contrario, para el mínimo será 0. La definición del pin deberá ser analógica y el modo estará definido como salida. De aquí es donde obtenemos el nombre de *PWM*, que ya explicamos en el capítulo 2.

analogRead()

Con esta función podemos leer los valores analógicos de un pin. Su declaración es analógica y puede ser tanto de entrada como de salida. Un ejemplo, `analogRead(pin);` Los valores que obtendremos estarán comprendidos entre 0 – 1023.

delay()

Es un retardo que hace esperar al microcontrolador un tiempo dado por el número especificado en milisegundos. Su declaración es `delay(50);` Significa que esperará 50 milisegundos antes de ejecutar la siguiente instrucción.

Serial.begin(valor)

Establece la velocidad de transmisión de datos en bits por segundo (baudios) para la transmisión de datos serie. En nuestro caso, la comunicación se ha realizado para 9600 baudios.

Serial.print(valor)

Imprime los datos al puerto serie como texto legible ASCII. Los datos *float* son impresos por defecto con dos decimales. *Serial.print()* no añade "*Enter*" ni nueva línea.

Serial.println(valor)

Imprime los datos al puerto serie como texto legible ASCII seguido por carácter "*Enter*" ('\r') y un carácter de nueva línea ('\n').

Capítulo 4
Funcionamiento

Ahora en este nuevo capítulo, mostraremos algunas de las capturas de pantalla que podemos recoger de la aplicación y explicaremos los pasos a seguir y apartados más importantes de los que se compone.

4.1 Instalación

La instalación de la aplicación es muy sencilla. Accedemos a la carpeta *download* de nuestro dispositivo. Allí, tendremos un archivo con el nombre de nuestra aplicación (ArduinoYun) con la extensión .apk como vemos en la ilustración 30.

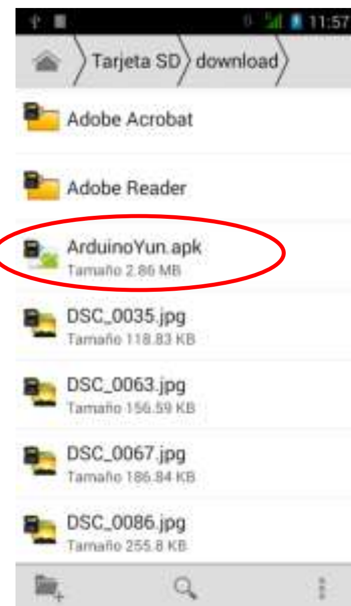


Ilustración 30: Archivo .apk

Procedemos a instalarla en nuestro teléfono marcando la casilla de "instalación de orígenes desconocidos" ya que no ha sido descargada desde la Play Store que es donde Google garantiza la seguridad de la aplicación. Ilustración 31.

Posteriormente, aceptamos los permisos (como bien recordamos, se declararon en el fichero *AndroidManifest.xml* del IDE) que la aplicación necesitará para funcionar correctamente. Ilustración 32.

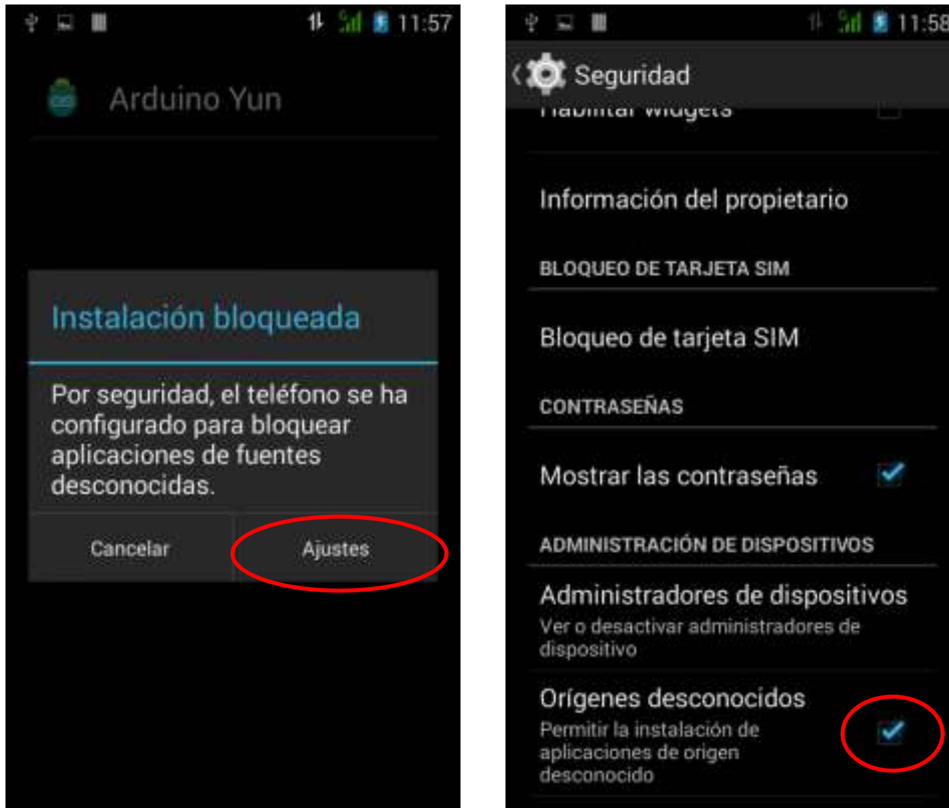


Ilustración 31: Seguridad en la instalación de aplicaciones

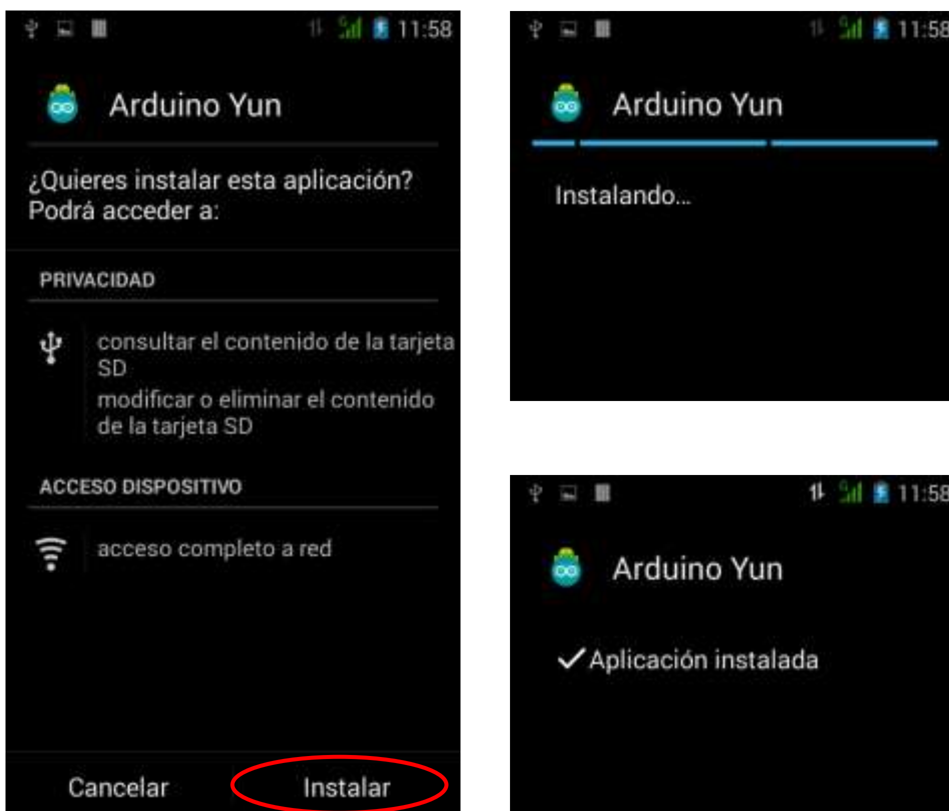


Ilustración 32: Permisos de la aplicación

4.2 Acceso a la aplicación

Una vez instalada la aplicación, se creará un icono en la pantalla principal. También podremos colocarlo en la vista de inicio como acceso rápido. Ambos casos podemos observarlos a través de la ilustración 33.

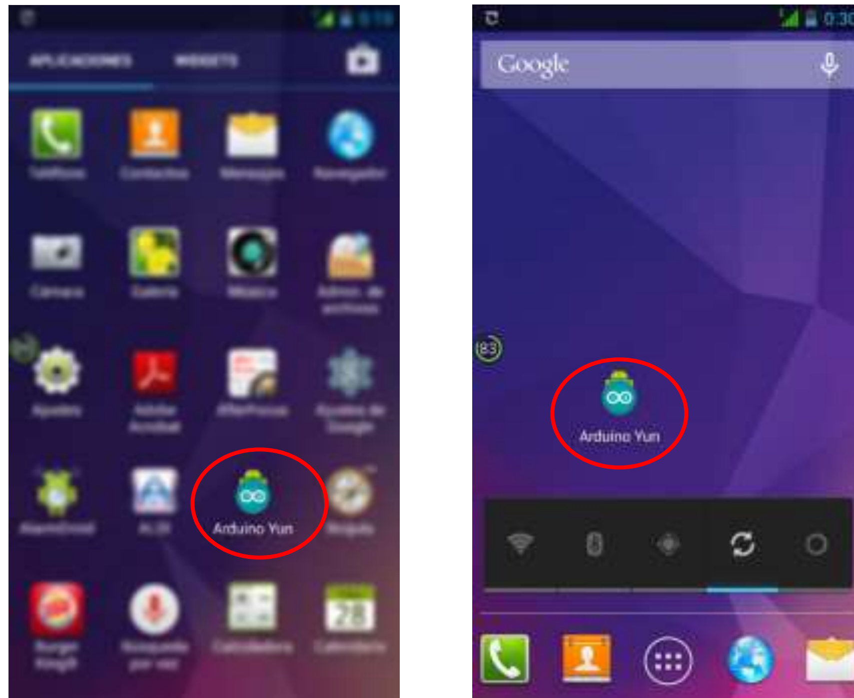


Ilustración 33: Icono de acceso a la aplicación

Una vez pulsado el icono accedemos a la aplicación dónde aparecerá, en primer lugar, una vista temporal como la representada en la ilustración 34 que nos muestra el icono de la aplicación y el nombre del desarrollador de ésta. Seguidamente nos adentramos en la propia aplicación donde podemos interaccionar con ella.

Ilustración 34: Imagen de lanzamiento de la aplicación

En primer lugar visualizamos la pantalla de Ajustes, en la cual configuraremos los pines como digitales, analógicos o no conectados. También encontramos un campo de texto donde poder introducir la dirección IP donde esté conectado nuestro Arduino y el puerto por el que accederemos a través del router. Por otra parte, incluye dos

botones que nos darán la posibilidad de conectarnos a los datos anteriores y guardar los ajustes de los pines para que al volver abrir nuestra aplicación, permanezcan configurados. Destacar que en cualquier vista, podemos acceder al menú lateral de navegación para acceder a otras opciones.

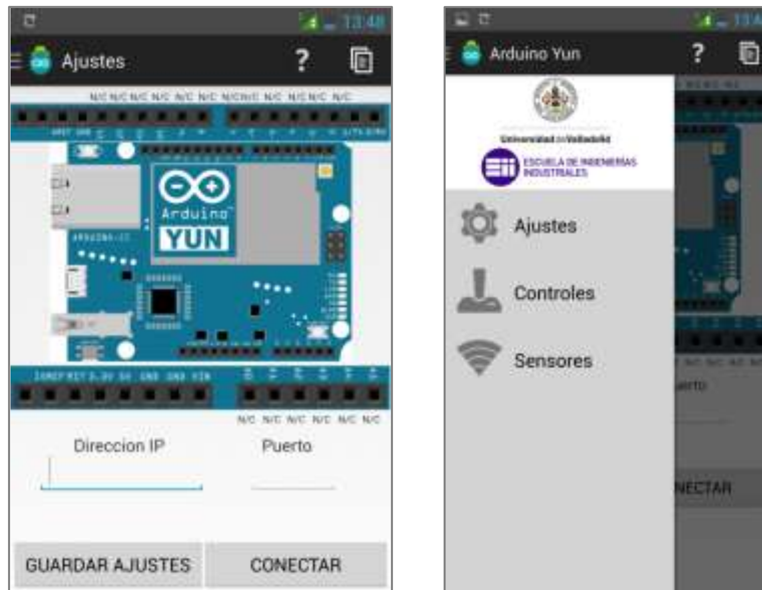


Ilustración 35: Ajustes y menú lateral de la aplicación

Como recoge la ilustración 35, a través del menú lateral, accedemos a las distintas opciones por las que navegar sobre la aplicación. La vista en detalle se muestra por medio de la ilustración 36. Este fragmento se dedica a los controles que podemos realizar sobre la tarjeta de Arduino y sus controles se habilitarán en función de las opciones de configuración que hay marcado el usuario en los Ajustes. Además podemos actualizar los estados de los botones mediante el botón “Actualizar”. Otra forma de acceder a la ventana “Controles”, se realiza mediante el botón “Guardar Ajustes” de la vista “Ajustes”

La ilustración 37, ahora nos muestra la interfaz “Sensores” donde podemos leer los pines analógicos de la placa Arduino. También es necesario configurar los pines desde “Ajustes” para realizar la lectura en la nueva vista. Otro aspecto a destacar, es el botón de lectura de sensores, que actualiza la barra de progreso en el momento.

Se ha decidido incluir este botón para no sobrecargar la memoria del sistema y que esté continuamente enviando peticiones a Arduino para saber el estado de los sensores en cada instante. Así el programa será más fluido y no haremos trabajar de forma excesiva al microcontrolador.

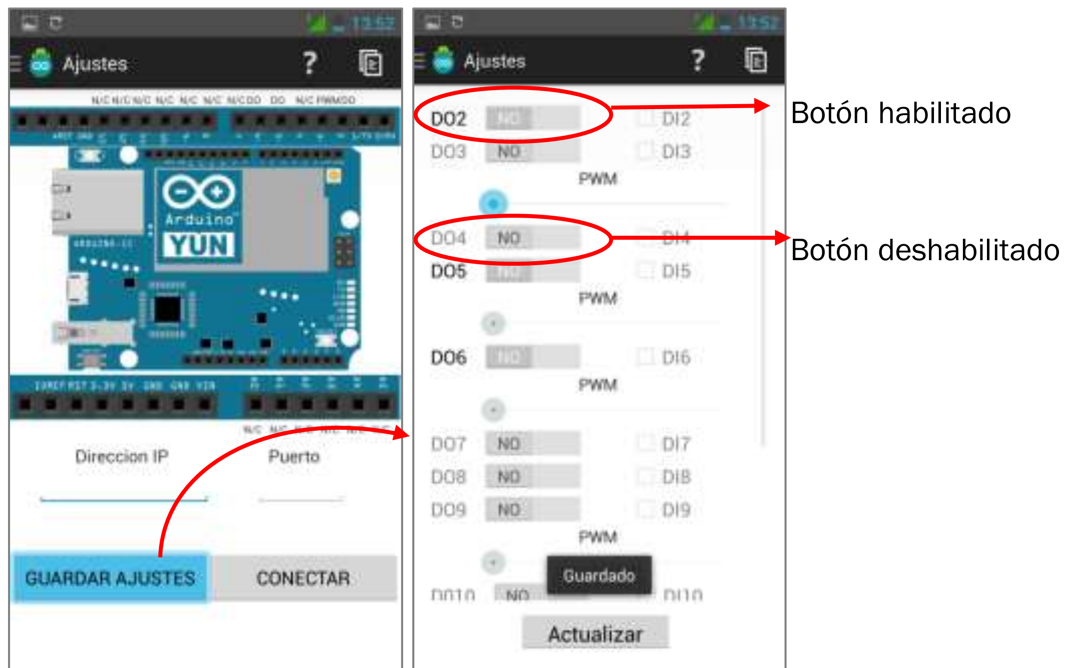


Ilustración 36: Vista “Controles” de la aplicación

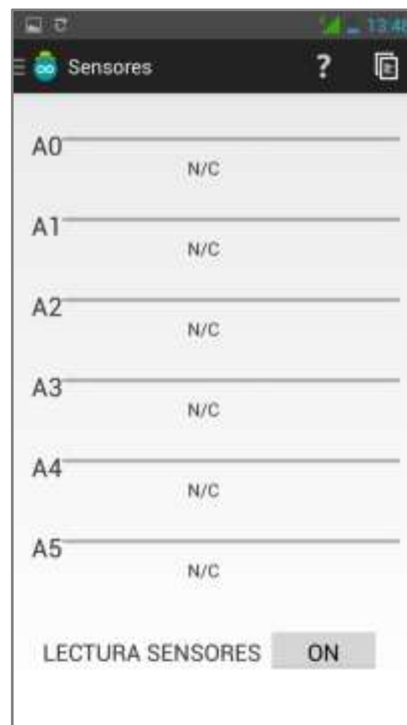


Ilustración 37: Vista "Sensores" de la aplicación

Otros recursos adicionales implantados en la aplicación, se encuentran en el *ActionBar*, representados a través de dos iconos. Ilustración 38.

Ilustración 38: Iconos en la *ActionBar*

La imagen de la izquierda nos ofrece una ayuda rápida sobre el manejo de la aplicación, con breves explicaciones de cada una de las pantallas. En la ilustración 39 vemos cada una de ellas.

Ilustración 39: Pantallas de ayuda de la aplicación

En cuanto a la ilustración 38, la imagen de la derecha, nos permite acceder a la carpeta del almacenamiento externo de la tarjeta SD donde se almacena la configuración de los pines.

Al pulsar sobre el icono mencionado, se muestra una ventana emergente con las posibles opciones con las que abrir el directorio. Nosotros hemos elegido el explorador de archivos (ES File Explorer) como vemos en la ilustración 40.

Desde este directorio, podemos cambiar la configuración de los pines de acuerdo a la nomenclatura que se exige en la ayuda. Es un camino paralelo que el usuario puede optar para obtener los mismos resultados.

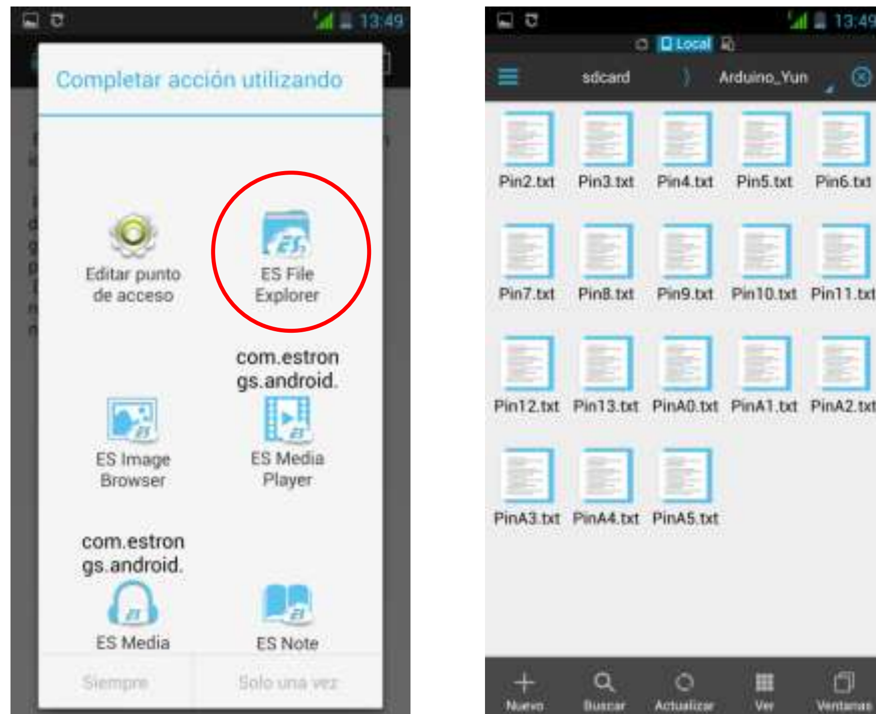


Ilustración 40: Almacenamiento externo con la configuración de los pines

Capítulo 5
Caso práctico

Con este nuevo capítulo, intentamos aclarar aún más el servicio que nos ofrece nuestra aplicación e implementarlo en un caso práctico. Mostraremos el manejo de Arduino a través de unos sencillos montajes, el circuito de conexión y los requisitos necesarios para cumplir dicha práctica.

5.1 Esquema general

El siguiente esquema de la ilustración 41 nos muestra la conexión cableada a nuestro Arduino Yun con diferentes componentes que utilizamos en nuestro caso práctico para visualizar el uso de nuestra aplicación.

5.2 Componentes electrónicos

Los componentes utilizados en el caso práctico son 5 leds, simulando el encendido y apagado de luces en el exterior y en el interior de la vivienda. Además incorporamos un servo que nos permite simular la apertura de una puerta de garaje de manera gradual o instantánea. En cuanto a los sensores, se ha incorporado un LDR que nos permite averiguar el grado de luminosidad que incide sobre el lateral de la casa, ya que es la posición donde se ha instalado.

En cuanto a la alimentación, usaremos una batería externa de tal forma que no sobrepase el límite de los 5 Voltios. Se alimentará mediante la entrada Micro USB.

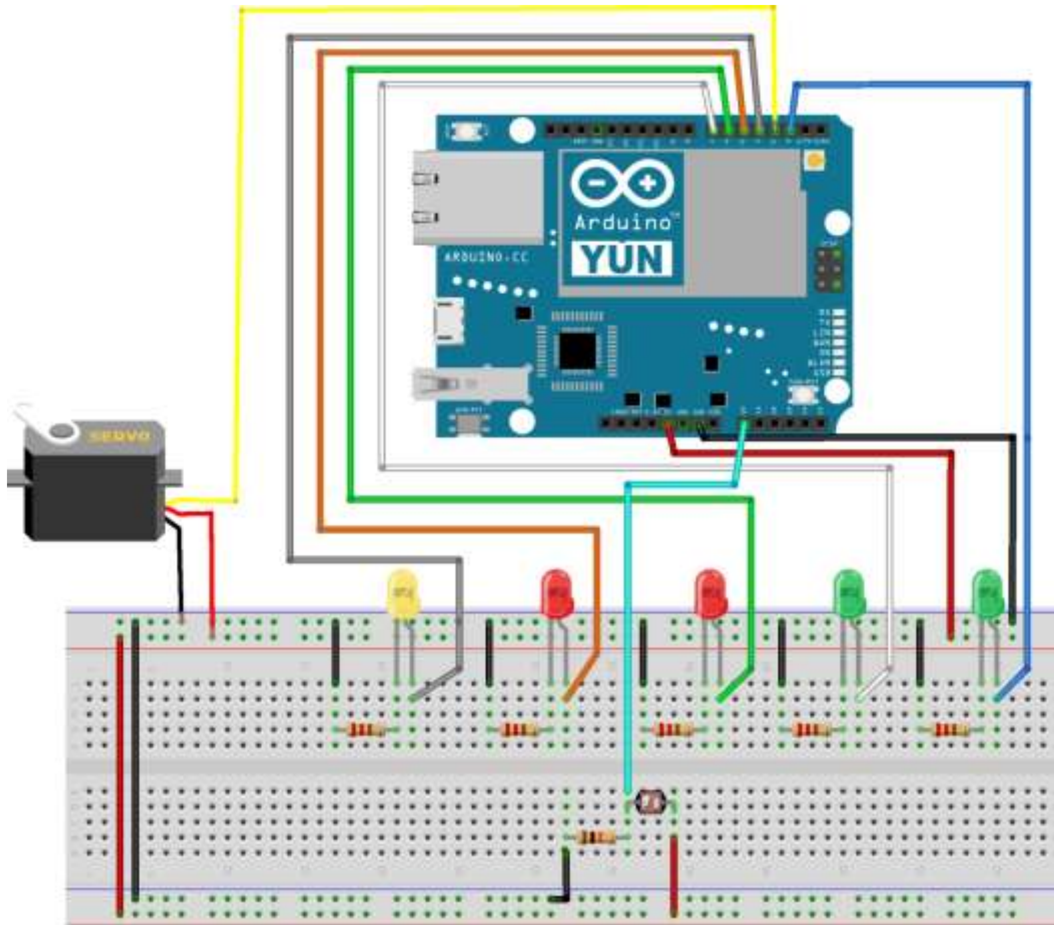


Ilustración 41: Esquema del cableado del caso práctico

5.2.1 Leds

Un led (*Light-Emitting Diode* o diodo emisor de luz) es un diodo que emite luz. Es un componente pasivo que cuando se le aplica una tensión entre sus dos terminales, permite que los electrones lo recorran a la vez que libera energía en forma de fotones, es decir, emite luz. La luz emitida no tiene por qué ser visible, podemos encontrar leds que emiten en infrarrojo hasta la franja ultravioleta, pasando por el espectro visible. Los más comunes son los rojos, amarillos o verdes.

Para aumentar su luminosidad, en su interior se encuentra una cavidad reflectora y su exterior está recubierto por una resina epoxi que hace de lente a la vez que protege el conjunto. Este recubrimiento puede ser tintado del mismo color que la luz que emitirá el diodo o transparente.

Los leds deben utilizarse respetando la polaridad. Tiene dos terminales diferentes, denominados ánodo y cátodo y deben conectarse de manera que el ánodo se una al borne positivo y el cátodo al negativo.

Para diferenciar estas dos patillas en un led, los fabricantes marcan el cátodo con la patilla más corta o la capsula rebajada por el lado negativo del diodo.

Nuestro caso práctico incorpora 5 leds, 2 de ellos rojos, otros dos verdes y uno amarillo, como vemos en la ilustración 42. A la hora de hacer las conexiones es importante leer la hoja de características del fabricante, aunque de manera genérica podemos decir que funcionan con una intensidad de 10 a 20 mA, dependiendo de si son de baja o alta intensidad y respecto a las caídas de tensión en función del color emitido. La Tabla 4 nos enseña una aproximación de estas caídas de tensión.

Ilustración 42: Leds

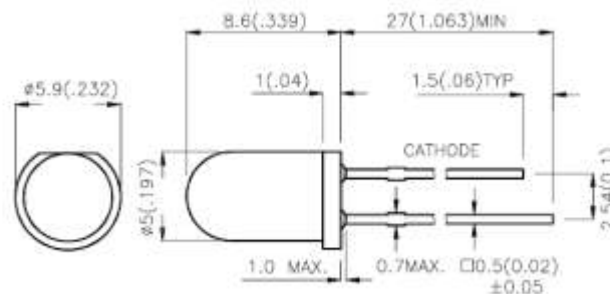


Ilustración 43: Dimensiones de un led

Color	Caída en voltios
Rojo	1.6 – 2.03
Naranja	2.03 - 2.10
Amarillo	2.1 – 2.3
Verde	1.9 – 3.7
Azul	2.47 – 3.7
Blanco	3.5
Infrarrojo	<1.63
Ultravioleta	3.1 – 4.3

Tabla 4: Caídas de tensión de los LEDs en función de color

Sabiendo que la salida de Arduino es de 5 Voltios y que puede suministrar una corriente de 40 mA, debemos limitar de alguna manera la corriente que recorrerá el diodo con tal de no dañarlo. Para ello utilizaremos la ley de Ohm y mediante los datos de la hoja de características, ilustración 44, realizaremos los cálculos necesarios para un cableado correcto.

$$V = I \cdot \mathrm{R} \qquad\qquad R = \frac{(V_{arduino} - \mathrm{V}_{led})}{\mathrm{I}}$$

Led rojo

Caída de tensión= 2 Voltios aproximadamente.

$$R = \frac{(5\mathrm{V} - 2\mathrm{V})}{20\ \mathrm{mA}} = 150\ \Omega$$

Por lo tanto, la resistencia que utilizaremos será como mínimo de 150Ω, pero podemos usar de 220Ω que son muy comunes.

Led verde

Caída de tensión= 3,5 Voltios aproximadamente.

$$R = \frac{(5\mathrm{V} - 3{,}5\mathrm{V})}{20\ \mathrm{mA}} = 75\ \Omega$$

La resistencia que usaremos, será de 220Ω igual que en el caso anterior, ya que será la que utilicemos para el resto de leds. En este caso, la luminosidad que nos aportará el led verde será menor que en el rojo y amarillo, ya que al poner una resistencia mucho mayor que la calculada, nos restará intensidad y el diodo emitirá menos luz.

Led amarillo

Caída de tensión= 2 Voltios aproximadamente.

$$R = \frac{(5\mathrm{V} - 2\mathrm{V})}{20\ \mathrm{mA}} = 150\ \Omega$$

Al igual que en el caso del led rojo, colocaremos una resistencia de 220Ω.

5.2.2 Servomotores

Los servomotores son unos motores que junto a un conjunto de elementos que lo acompañan (un controlador, un motor de corriente continua, una caja reductora y un potenciómetro), son capaces de posicionarse en una ubicación determinada dentro de su rango de operación y mantenerse estable en dicha posición dependiendo de la señal recibida. Disponen de un terminal donde enviar una señal que es transformada en una orden acerca de cómo debe moverse y posicionarse el motor.

Su funcionamiento básico es el siguiente: El motor se encuentra conectado a la caja de engranajes que se encarga de transmitir el movimiento reducido al brazo del servo. Ésta caja de engranajes a su vez mueve el potenciómetro que actúa de sensor cuando se recibe una señal de posición y la transforma en voltaje equivalente y lo compara con el voltaje obtenido por el potenciómetro. Dependiendo de las diferencias entre ellos, el motor gira para equipararlos.

El movimiento de brazo del servomotor se indica mediante PWM. Cada posición del motor viene dado por un pulso diferente. Si el pulso es de 1ms, indica que se debe colocar a 0º. Si el pulso es de 2ms a 180º y a partir de estos dos valores se extrapolan el resto. Normalmente el ciclo completo de pulso es de 20 ms.

Como en la ilustración 44, dispone de 3 conexiones. Un cable se conecta a masa, el segundo a +5V y el tercero al pin analógico PWM. Habitualmente el fabricante indica que cable corresponde cada conexión. También, mediante la colorimetría, a partir del negro, rojo y blanco respectivamente podemos establecer la conexión.

Ilustración 44: Servomotor

5.2.3 Fotorresistencias

Una LDR (*Light-Dependent Resistor* o resistencia dependiente de la luz) es un componente cuya resistencia varía con la intensidad de la luz. Concretamente su resistencia disminuye cuanto mayor es la intensidad de luz recibida. Son conocidas también como fotorresistencias o sensores de luz. En la ilustración 45 vemos un ejemplo.

Ilustración 45: LDR

Las fotorresistencias son elementos no polarizados, es decir, que no tenemos que preocuparnos de cuáles son los terminales que debemos conectar, ya que actúan de igual manera que la resistencias.

Para trabajar con los LDR, normalmente se utiliza un divisor de tensión y como punto de medida, se toma precisamente la unión de la fotorresistencia con la resistencia fija según el esquema mostrado en la ilustración 46.

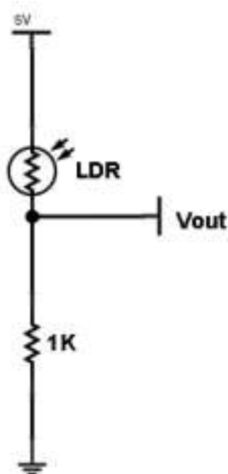


Ilustración 46: Divisor de tensión

Si utilizamos el LDR como resistencia superior, tendremos la tensión máxima cuando esté completamente iluminado, ya que se comportará prácticamente como un circuito abierto.

Dependiendo de la fotorresistencia que tengamos disponible, puede tener la resistencia de divisor de tensión incorporada o no, en caso de no tenerla utilizaremos una resistencia externa de 10KΩ. La entrada de datos hacia Arduino se realiza sobre la caída de tensión de la resistencia. Cuando haya luz, tendremos que la resistencia interna del LDR será de unos cientos de ohmios con lo que la lectura será cercana a los 5 Voltios (al ser entrada analógica, debería ser un valor cercano a los 1023), mientras que cuando no haya luz, la resistencia interna aumentará y alcanzará valores de megaohmios, mucho mayor de los 10K de la resistencia de referencia. Esto nos indica que tendremos una lectura cerca de los 0 Voltios.

Si colocamos la LDR en la parte inferior, obtenemos resultados inversos en cuanto al sentido de menor luz, mayor voltaje.

Capítulo 6
Conclusiones y trabajos futuros

Con este último capítulo, resumiremos la labor realizada a lo largo del trabajo fin de grado. Numeraremos los objetivos cumplidos, analizaremos las principales aportaciones y trabajos futuros que puedan mejorar el presente proyecto. Por último, añadiremos una valoración personal.

6.1 Objetivos cumplidos

Desde el comienzo del proyecto, fueron necesarios unos requisitos, los cuáles debíamos alcanzar para lograr el éxito. Los objetivos logrados han sido:

- ✓ Conocer las principales características de los lenguajes Java y C.
- ✓ Conocer las principales características de Android y Arduino.
- ✓ Estudiar el entorno de desarrollo de Android y Arduino.
- ✓ Desarrollar la aplicación para Android.
- ✓ Programar el código para Arduino.
- ✓ Establecer la conexión y comunicación entre las dos plataformas.
- ✓ Envío y respuesta de datos para el control remoto.
- ✓ Lectura de sensores de la placa Arduino.
- ✓ Aplicación práctica.

Con este desarrollo hemos alcanzado todos los objetivos que habíamos fijado al comienzo del proyecto. El estudio en detalle sobre todo del lenguaje Java, muy complejo en algunos aspectos, nos ha ayudado a realizar mejoras en la aplicación, mejorando estilos y diseños de aplicaciones

obsoletas hasta conseguir interfaces actualizadas a día de hoy, para poder interaccionar mejor con el usuario y reaccionar ante cualquier nueva actualización.

Respecto a la programación en C, conocida más en profundidad que el lenguaje Java, nos ha permitido realizar un código sencillo y eficaz para un buen rendimiento de nuestra placa Arduino.

En cuanto a la comunicación entre Android y Arduino responden ambos, correctamente, a las peticiones para recibir y enviar datos desde cada terminal. El único detalle mejorable podríamos atribuirlo al tiempo de respuesta desde que un dispositivo envía la información y el otro lo recibe. En el caso de Arduino es debido a que después de cada petición, incorporamos un retardo de 50 ms para evitar la sobrecarga de trabajo del procesador. Respecto a Android, la implementación de los numerosos hilos secundarios correspondientes a cada pin que puede enviar o recibir datos retrasa el tiempo de ejecución del programa.

6.2 Mejoras y trabajos futuros

Una vez terminado el proyecto, en este apartado describiremos algunos detalles que podrían aplicarse a nuestro trabajo.

En primer lugar, sobre la aplicación en Android, una línea futura de trabajo sería trabajar sobre las notificaciones, de tal manera que cuando algún sensor alcance un valor determinado, se notifique en la barra de notificaciones o bien, avisarnos mediante un correo o aviso en nuestras redes sociales.

El segundo trabajo sobre el que poder tratar, sería incluir una nueva interfaz en la aplicación de Android con diferentes opciones ya personalizadas para trabajar algún aspecto en detalle en unas prácticas de laboratorio.

En cuanto a mejoras, podríamos disminuir el pequeño retardo de 50 ms que tenemos entre que solicitamos la acción y ésta se realiza sin sobrecalentar nuestro Arduino Yun. Por el contrario, podríamos disminuir el número de hilos secundarios que no se vayan a utilizar en la programación de Android para recibir la información de Arduino de manera más rápida.

Bibliografía

Monografías

[1] **ABLESON, Frank**. *Guía para desarrolladores*. 2ª Edición. Anaya, 2011

[2] **FRIESEN, Jeff**. *Java para desarrollo Android*. Edición 2011. Anaya, 2011

[3] **MONTERO MIGUEL, Roberto**. *Desarrollo de aplicaciones para Android*. Edición 2014. RA-MA, 2014

[4] **RIBAS LEQUERICA, Joan**. *Desarrollo de aplicaciones para Android*. Edición 2015. Anaya, 2015

[5] **RIBAS LEQUERICA, Joan**. *Arduino Práctico*. Edición 2014. Anaya, 2014

[6] **TOMAS GIRONES, Jesús**. *El gran libro de Android*. 4ª Edición. Marcombo, 2011

Publicaciones electrónicas

[6] *20 conceptos de Android* [en línea]. El Android Libre. http://www.elandroidelibre.com/?s=android+20+conceptos

[7] *Arduino Yun* [en línea]. Arduino. http://www.arduino.cc/en/Guide/ArduinoYun

[8] *Arquitectura de Android* [en línea]. Androideity.
http://www.androideity.com/2011/07/04/arquitectura-de-android

[9] *Ciclo de vida de un fragmento* [en línea]. Edu4java.
http://www.edu4java.com/es/androidlibro/gestionar-ciclo-de-vida-fragmento-android.html

[10] *Comunicación con fragmentos* [en línea]. MundoGeek.
http://www.mundogeek.net/android/fragmentos/comunicacion.htm

[11] Data storage [en línea]. developer.android.
http://developer.android.com/guide/topics/data/data-storage.html

[12] *Fragmentos* [en línea]. developer.android.
http://developer.android.com/guide/components/fragments.html

[13] Tareas asíncronas [en línea]. Android curso.
http://www.androidcurso.com/index.phptutoriales-android-fundamentos/.../365-ejecutar-una-tarea-en-un-nuevo-hilo-con-asyntask

[14] *Navigation Drawer* [en línea]. Creando Android.
http://www.creandoandroid.es/implementar-navigation-drawer-menu-lateral

Anexo A
Guía de instalación de Eclipse

Esta guía de instalación nos ayudará a instalar el entorno de desarrollo de Eclipse para poder desarrollar las aplicaciones que implementaremos en nuestros teléfonos. Se detallarán los requisitos mínimos para completar la instalación y los archivos descargables necesarios.

En primer lugar, descargamos el paquete del IDE. Existen dos formas diferentes. La primera es acceder a la página web *www.eclipse.org/downloads/* donde nos descargaremos el paquete de "*Eclipse IDE for Java Developers*" para 32 bits o 64 bits dependiendo del sistema operativo donde se vaya a ejecutar. La segunda a través de la página de Google *http://developer.android.com/sdk/index.html* en la que incluye el nuevo IDE Android Studio, Java y el JDK de Java y el SDK de Android. Este segundo método es instantáneo, pero es necesario entender cada paso que se realiza.

Ilustración 47: Paquete de descarga del entorno de desarrollo

Una vez descargado, descomprimimos el fichero y creamos un entorno de trabajo que posteriormente asociaremos a Eclipse para guardar todos nuestros proyectos.

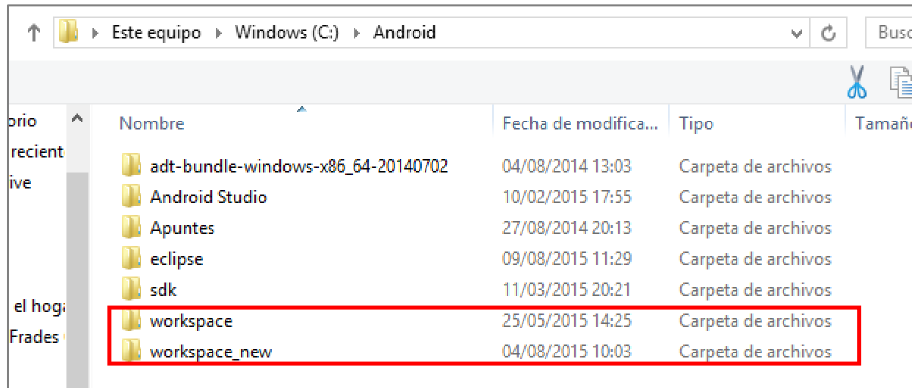


Ilustración 48: Espacio de trabajo de Eclipse

Antes de iniciar Eclipse debemos instalar la maquina virtual de Java. Para comprobar si ya está instalada en nuestro ordenador o queremos descargarla accederemos al siguiente enlace, *java.com/es/download/.* Pulsamos sobre el recuadro de “descarga gratuita” para que comience la instalación y una vez finalizada ejecutamos el archivo. Ya tendremos Java instalado.

Ilustración 49: Instalación y comprobación de Java

También es recomendable instalar el JDK para evitar posibles fallos futuros. Este paquete sirve para desarrollar software Java y disponer de herramientas adicionales. El link de descarga es *www.oracle.com/technetwork/java/javase/downloads/index.html* y como en casos anteriores, descargaremos la última versión disponible. Pulsaremos sobre el recuadro “*Download*” que se encuentra debajo de las siglas JDK, como indica la ilustración 44. Aceptamos los términos de licencia y descargamos el fichero que corresponda con nuestro sistema operativo. Una vez terminada la descarga ejecutamos el archivo y ya tendremos listo nuestro nuevo paquete Java.

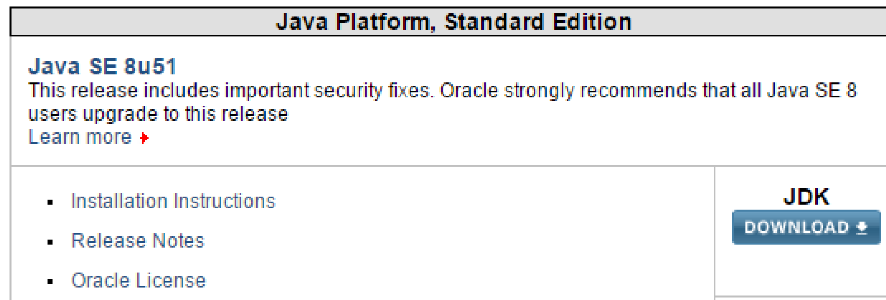


Ilustración 50: Descarga del paquete JDK de Java

Ahora sí, podremos abrir Eclipse. Durante el inicio aparecerá una ventana donde indicaremos el espacio de trabajo creado en pasos anteriores. A continuación, se abrirá la ventana del entorno de desarrollo que vimos en el capítulo 1.

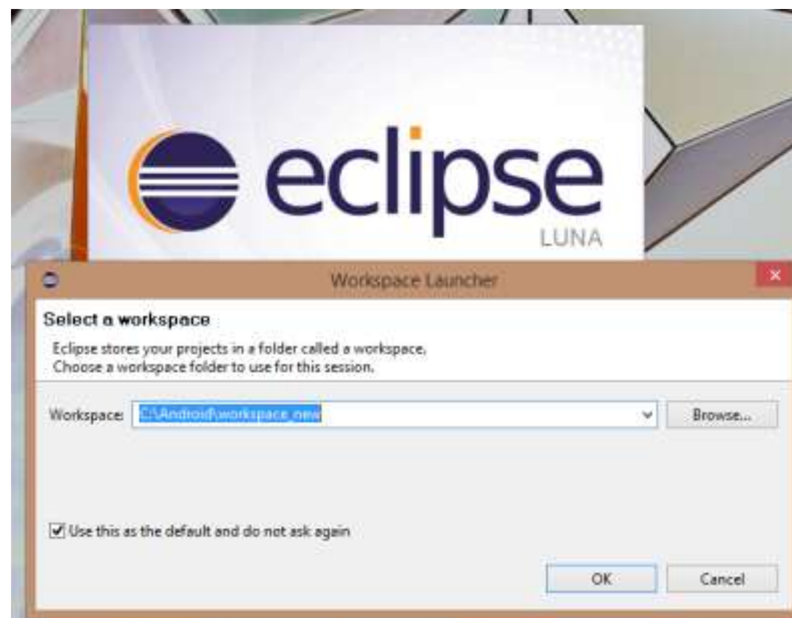


Ilustración 51: Directorio del espacio de trabajo de Eclipse

Una vez instalado Eclipse, Java y el JDK de Java, vamos a instalar el SDK de Android que encontraremos en el segundo enlace de este anexo. Se abrirá un instalador para guiarnos en la instalación y guardar en un directorio todos los ficheros que se descarguen. Ilustración 46.

Cuando haya terminado, se abrirá una ventana con todos los recursos de Android descargables. Instalaremos los paquetes (recomendable instalar todos para futuras ocasiones) y cerraremos la ventana. Ilustración 47.

SDK Tools Only

If you prefer to use a different IDE or run the tools from the command line or with build scripts, you can instead download the stand-alone Android SDK Tools. These packages provide the basic SDK tools for app development, without an IDE. Also see the SDK tools release notes.

Platform	Package	Size	SHA-1 Checksum
Windows	installer_r24.3.3-windows.exe (Recommended)	139463749 bytes	bbdae40a7665e55b3cdb1fbae865986e6cd3df14
	android-sdk_r24.3.3-windows.zip	187480692 bytes	b6a4899efbf20fc593042f1515446c6630ba502e
Mac OS X	android-sdk_r24.3.3-macosx.zip	98330824 bytes	41f0f3e76d6868018740e654aefb04fd765c357d
Linux	android-sdk_r24.3.3-linux.tgz	309109716 bytes	cd4cab76c2a3d926b3495c2fec56c831ba77d0d0

Ilustración 52: Enlace de descarga SDK

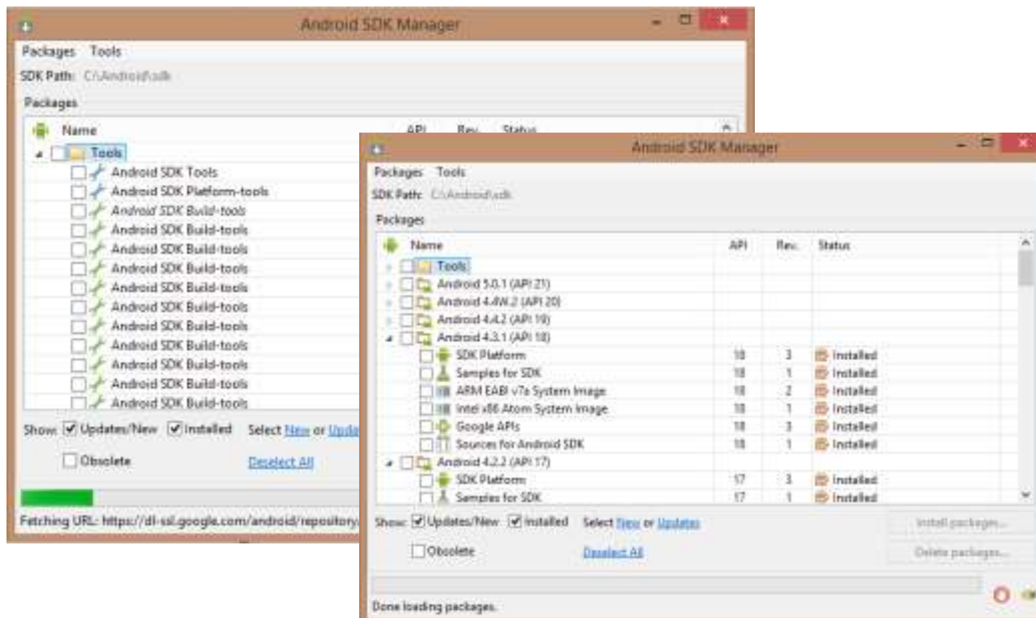


Ilustración 53: Recursos SDK Android

Ahora tenemos por un lado todos los recursos de Android incluyendo APIs, librerías etc. y el entorno de desarrollo Eclipse. Para vincularlos nos iremos al IDE y en el menú *"Help"* elegimos *"Install New Software"*. Seguidamente pulsaremos el botón *Add* como muestra la ilustración 48.

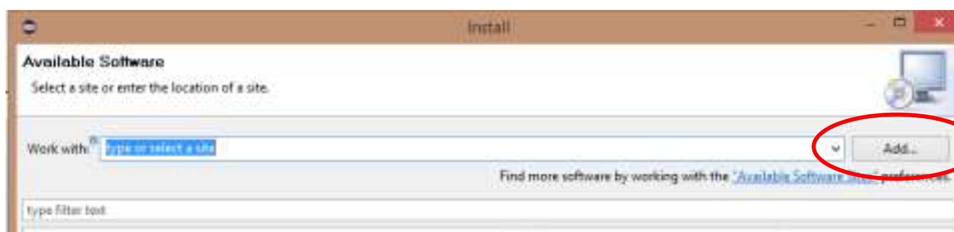


Ilustración 54: Instalación del software

A continuación escribiremos la localización en la que se encuentran los recursos descargados del SDK de Android. Finalizamos la instalación, reiniciaremos Eclipse y ya podremos trabajar en él.

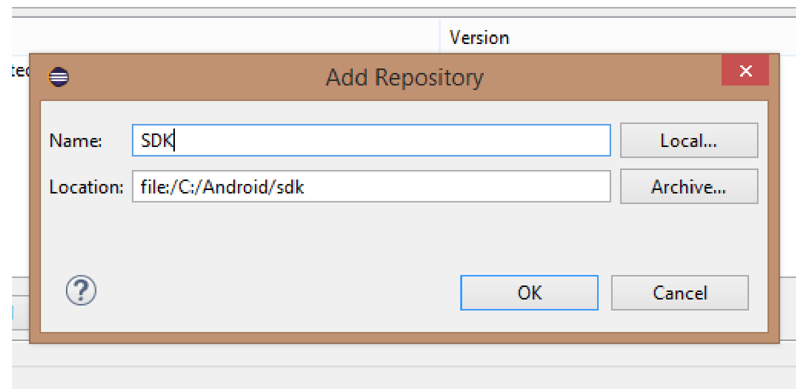


Ilustración 55: Localización del SDK de Android

Anexo B
Guía de instalación de Arduino

En el anexo B incluimos una guía de instalación del entorno de Arduino, donde aprenderemos paso a paso a configurar de forma correcta nuestro Arduino Yun por USB al ordenador, así como a instalar el software que nos permitirá desarrollar el código de nuestro programa para su posterior carga a la placa.

Descargaremos el entorno de desarrollo de Arduino desde su página web a través del enlace *http://arduino.cc/en/Main/Software*

Ilustración 56: Descarga del software

Descomprimimos el fichero y abrimos el ejecutable *.exe* que situaremos en el escritorio para acceder en ocasiones futuras al programa. Realizaremos el paso anterior ya que no necesita instalación previa.

En segundo lugar, conectaremos nuestro microcontrolador a un puerto USB del ordenador y esperaremos a completar la instalación de los *drivers.* Si el ordenador no es capaz de actualizarlo automáticamente, lo haremos

manualmente, dirigiéndonos a la ruta Equipo > Propiedades > Administrador de dispositivos. Si aparece un símbolo de un triangulo amarillo junto al puerto, significa que no se han instalado los *drivers*. En esta ventana también podemos ver el puerto al que se ha conectado nuestro Arduino Yun para luego declararlo en el entorno de desarrollo.

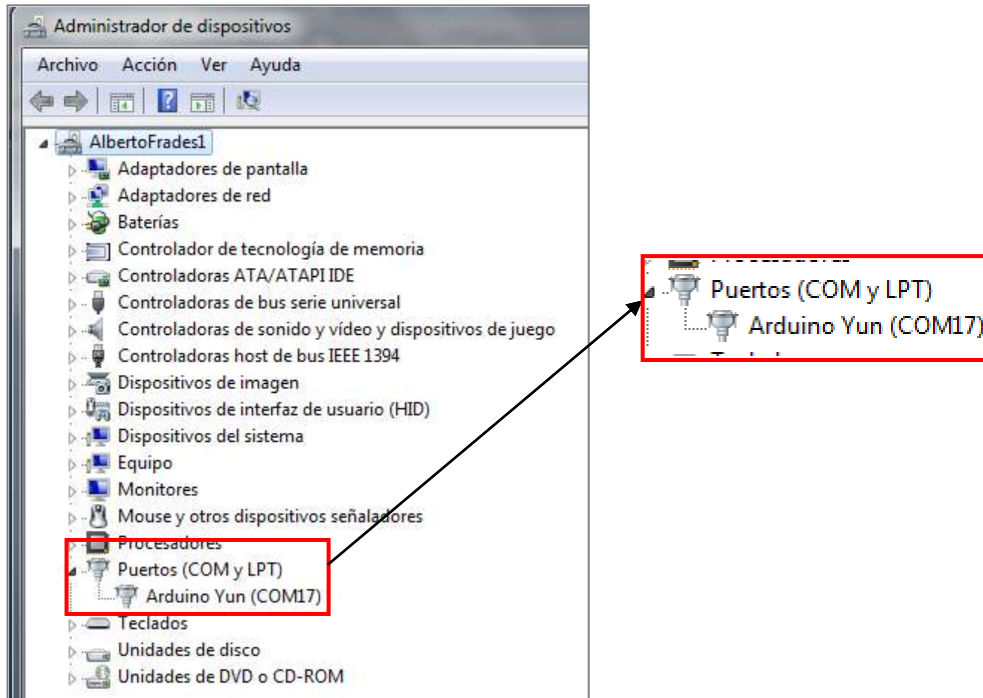


Ilustración 57: Administrador de dispositivos

Anexo C
Contenido del CD-ROM

Todos los ficheros generados a lo largo del desarrollo de este proyecto han sido incluidos en el CD-ROM que acompaña esta memoria. A continuación, mostramos una breve descripción del contenido:

- Memoria: Archivo en formato .pdf
- "Información adicional": Código fuente de la aplicación en Android.
  Código fuente de Arduino.