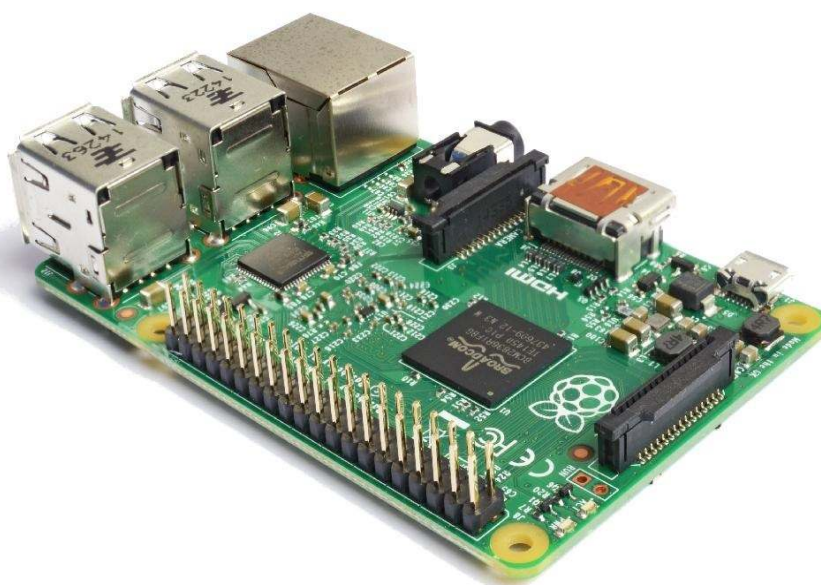
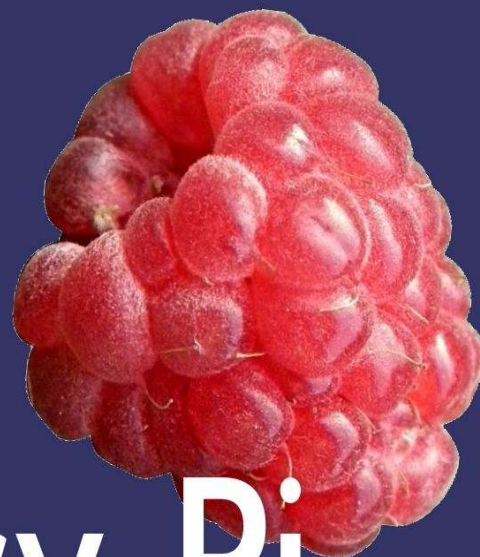


Taller de Raspberry Pi

Tercera edición



Francisco Moya Fernández
Universidad de Castilla-La Mancha
Escuela de Ingeniería Industrial

Tabla de contenido

Resumen	1.1
Introducción	1.2
La Raspberry Pi	1.2.1
El sistema GNU/Linux	1.2.2
Los periféricos de la RPi	1.3
Entradas y salidas digitales	1.3.1
Comunicaciones I2C	1.3.2
Comunicaciones SPI	1.3.3
Comunicación con UART	1.3.4
Comunicaciones en red	1.3.5
Desarrollo en C	1.4
Programando los periféricos	1.4.1
Entradas y salidas digitales	1.4.1.1
Comunicaciones I2C	1.4.1.2
Comunicaciones SPI	1.4.1.3
Comunicaciones en red	1.4.1.4
Arquitectura de software	1.4.2
Tratamiento de errores en C	1.4.2.1
Programación orientada a objetos	1.4.2.2
La biblioteca reactor	1.4.2.3
Casos de estudio	1.4.3
Dispositivo MP3	1.4.3.1
Piano de juguete	1.4.3.2
Desarrollo en Python	1.5
Programando los periféricos	1.5.1
Entradas y salidas digitales	1.5.1.1
Comunicaciones I2C	1.5.1.2
Comunicaciones SPI	1.5.1.3
Comunicaciones en red	1.5.1.4
Arquitectura de software	1.5.2
Casos de estudio	1.5.2.1
Dispositivo MP3	1.5.2.1.1
Acelerómetro	1.5.2.1.2
Control de accesos	1.5.2.1.3
Apéndices	1.6
Nuestra personalización de Raspbian	1.6.1
Secuencia de arranque	1.6.2
Alimentación de Raspberry Pi	1.6.3
Configuración de Raspberry Pi	1.6.4

Taller de Raspberry Pi

Taller de Raspberry Pi por Francisco Moya se distribuye bajo la licencia [Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional License](#).



Documento creado a partir de la obra en <https://github.com/FranciscoMoya/rpi-workshop>.

Estamos en la tercera edición de nuestro *Taller de introducción a Raspberry Pi*. Las dos ediciones anteriores se celebraron con el mismo entusiasmo que ésta pero con mucho menos tiempo de preparación. Esta vez contamos con mejor documentación y sobre todo con un contenido mucho más enfocado a los resultados prácticos.

Raspberry Pi es un pequeño ordenador personal diseñado como una herramienta para el aprendizaje de programación de computadores y de diseño de sistemas electrónicos digitales. Desde un punto de vista completamente práctico veremos cómo configurarla y usarla para desarrollar programas y sistemas electrónicos propios. Se invertirá un esfuerzo considerable en comunicar la Raspberry Pi con el mundo físico, añadiendo sensores y actuadores de diversos tipos.

Hemos puesto mucha ilusión en esta renovación del taller. Esperamos que cubra tus expectativas, pero si no lo hace no dudes en hacernos llegar tus sugerencias.

Happy hacking!

Toledo, Tue Jan 17 2017 13:23:08 GMT+0000 (UTC)

Francisco Moya Fernández

Introducción

Un taller no aspira a dar unos conocimientos teóricos profundos, sino que tiene una orientación estrictamente práctica. Te enseñaremos lo que incluye la Raspberry Pi, cómo configurarla y cómo usarla para realizar tus propios proyectos. Se pretende que los ejemplos sean abordables incluso por alumnos de primer curso que ya hayan cursado la asignatura de *Informática*.

Una importante incorporación en esta edición del taller son las nociones de arquitectura software. Queremos que se usen las *Raspberry Pi* para abordar problemas reales de ingeniería. Y para eso es necesario que el software desarrollado sea considerablemente más evolucionado que lo que solemos ver en los *trabajos fin de grado*. Los ejemplos que abordaremos son sencillos, pero no triviales. Se proporcionarán plantillas de componentes reusables para construir sistemas relativamente sofisticados.

Info Todo el código que te entregamos con el taller puedes usarlo en tus propios trabajos y proyectos. Se distribuye bajo la [licencia pública de GNU](#), una licencia permisiva que te permite incluso modificar el software o explotar comercialmente tus proyectos. Solo hay una condición, los trabajos derivados solo se pueden distribuir bajo esta licencia.



Una limitación importante de este taller es que no tratamos con sistemas de tiempo real estricto pero no podemos hacer más en dos créditos. En un futuro próximo intentaremos ofrecer cursos complementarios de tiempo real y robótica con *Raspberry Pi*.

Kit del alumno

Este taller está concebido como una actividad de motivación *pro-bono*, sin ningún tipo de remuneración para el personal involucrado en el curso. El 100% del dinero recaudado en las matrículas se invierte en el material que se lleva el alumno. Las compras se realizan con meses de antelación, gracias a la colaboración de la Escuela de Ingeniería Industrial de Toledo, para poder aprovechar ofertas y proveedores extranjeros.

Cada edición del taller tiene su propia selección de componentes. En esta edición la selección ha sido la siguiente:

Componente

Raspberry Pi 3 modelo B
Caja Raspberry Pi B+
Fuente 5.1V 2.5A
Cables Dupont (40 + 40 + 40)
Conversor HDMI to VGA
LEDs, switches, resistencias, potenciómetros
Tarjeta microSD 8GB
Protoboard pequeña
Servo de engranajes metálicos
Cable USB-UART
Lector microUSB
Acelerómetro + giróscopo I2C
ADC SPI
Teclado matricial de membrana
Convertidor bidireccional 3.3V a 5V

Enlace

<http://es.farnell.com/raspberry-pi/raspberrypi-modb-1gb/raspberr>
<http://www.banggood.com/ABS-Case-For-Raspberry-Pi-B-Black->
<http://es.farnell.com/stontronics/t6090dv/psu-raspberry-pi-5v-2-5>
<http://www.banggood.com/120Pcs-20cm-Color-Breadboard-Jum>
<http://www.banggood.com/1080P-HDMI-Male-To-VGA-Female-A>
<http://www.banggood.com/Electronic-Parts-Component-Resistor>
<http://www.banggood.com/8GB-MicroSD-TF-Memory-Card-For-F>
http://www.banggood.com/10Pcs-8_5-x-5_5cm-White-400-Holes
<http://www.banggood.com/4-X-Towerpro-MG90S-Metal-Gear-RC>
<http://www.banggood.com/USB-to-RS232-TTL-Serial-FTDI-Chip>
http://www.banggood.com/Wholesale-Diamond-USB-2_0-Hi-Spe
<http://www.banggood.com/5Pcs-6DOF-MPU-6050-3-Axis-Gyro-f>
<http://www.banggood.com/CJMCU-1118-ADS1118-16-bit-ADC-A>
<http://www.banggood.com/5Pcs-4-x-3-Matrix-12-Key-Array-Mem>
<http://www.banggood.com/20Pcs-8-Channel-Logic-Level-Transla>

Preselección de componentes para esta edición.

Los enlaces a la derecha te llevarán al sitio del fabricante seleccionado. Los precios pueden variar ligeramente respecto al momento de compra. Por ejemplo, en marzo ya teníamos las Raspberry Pi 3 modelo B, los alimentadores originales y las tarjetas microSD. Hemos tenido problemas en el pasado con otro tipo de alimentadores más baratos y no hemos querido arriesgar.

Diferencias con ediciones pasadas

Todo es diferente. Las ediciones pasadas del taller dedicaban la mayor parte del tiempo a conseguir un entorno de desarrollo cómodo con la Raspberry Pi. Esto limitaba enormemente el tiempo que podíamos dedicar a hacer proyectos y, por tanto, la utilidad del taller.

En esta edición todo el material se proporcionará completamente configurado y listo para usarse y no será necesario ningún software en otro ordenador personal. El kit del alumno incluirá un conversor HDMI-VGA para poder usar los monitores del laboratorio y se conectará directamente el ratón y el teclado USB del puesto de laboratorio.

También hemos tratado los problemas de infraestructura que sufrimos en ediciones pasadas. En primer lugar hemos preparado el taller para que no sea necesario ningún tipo de comunicación con el exterior, ni soporte de aplicaciones externas tales como *Bonjour*. Proporcionamos todos los mecanismos de comunicación posibles previamente configurados pero no necesitaremos ninguno.

Warning Dada la radicalidad de los cambios que hemos hecho en el taller es posible que convoquemos una edición 3b en septiembre. El objetivo es reducir sensiblemente el precio de la matrícula eliminando del kit del alumno la Raspberry Pi 3 y los componentes ya incluidos en otras ediciones. De esta forma alumnos que ya han cursado ediciones pasadas del taller pueden actualizar su formación. En cualquier caso estas ediciones intermedias solo se producirán si hay demanda suficiente.

Otra diferencia importante es que en esta edición incorporamos un [nuevo sitio web](#) de soporte al curso. Queremos activar una comunidad de usuarios interesados alrededor de este sitio. Participa y haznos llegar tus sugerencias.

Estructura del manual

Este manual está dividido en tres partes:

- La primera parte introduce la Raspberry Pi, sus características, su historia, el sistema operativo que vamos a emplear, y el entorno de desarrollo.
- La segunda parte describe los diferentes componentes de la Raspberry Pi desde un punto de vista aislado. Se trata de que el alumno sepa cómo se programa cada componente y qué limitaciones tiene.
- Finalmente la última parte se dedica a temas de arquitectura software. Cómo construimos programas que tratan con múltiples fuentes de eventos heterogéneas. Cómo se organiza un programa complejo para que no sea imposible modificarlo.

Repositorios GitHub

La última versión del material del curso está disponible en todo momento en [GitHub](#) en los siguientes repositorios:

```
https://github.com/FranciscoMoya/rpi-doc.git
https://github.com/FranciscoMoya/rpi-src.git
```

El primer repositorio corresponde a la documentación del taller, a los archivos a partir de los cuales se genera este manual. A menos que quieras adaptarlo para otro fin es probable que prefieras descargarla de [gitbooks.io](#). Ten presente que el manual no se distribuye bajo la licencia de GNU, sino bajo la licencia [Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional](#).

El segundo repositorio corresponde al software de apoyo, que tienes preinstalado en tu *Raspberry Pi*. Para actualizarlo ejecuta un terminal de órdenes y escribe:

```
cd src
git pull -u
```

Organización de este manual

Este pequeño libro se ha dividido en tres partes:

- La primera parte se dedica al material de fundamentos e introducción necesario. Lo ideal sería que el alumno tuviera

ya al menos estos conocimientos al empezar el taller, pero dedicaremos el primer día a revisarlos.

- La segunda parte introduce los periféricos y la programación de los periféricos desde un punto de vista agnóstico respecto al lenguaje. Se utilizan herramientas de línea de órdenes para interactuar con ellos.
- La tercera parte se dedica a la programación de la Raspberry Pi. Se ofrecen dos versiones, una en C y otra en Python. Incluye material de fundamentos, ejemplos sencillos equivalentes a la segunda parte, y casos de estudio.

Obviamente esta organización está influida por aspectos prácticos. Dejamos para la segunda parte todo el material común que podemos, para no repetirlo en las terceras partes C y Python. En el taller vemos todo junto, presentando en paralelo versiones C y Python.

La Raspberry Pi

En el año 2006, un grupo del Computer Lab de la Universidad de Cambridge empezó a preocuparse por el nivel con el que llegaban los alumnos de secundaria a la Universidad. Por alguna razón los alumnos que tenían alguna exposición previa con tecnologías informáticas acumulaban conocimientos sobre aplicaciones concretas, en lugar de conocimientos sobre las propias tecnologías. Raspberry Pi surge como una iniciativa de bajo coste para promover la experimentación con la programación desde edades tempranas, aunque no por ello se trata de un mero juguete.



Raspberry Pi 3 modelo B. Foto de [raspberrypi.org](https://www.raspberrypi.org).

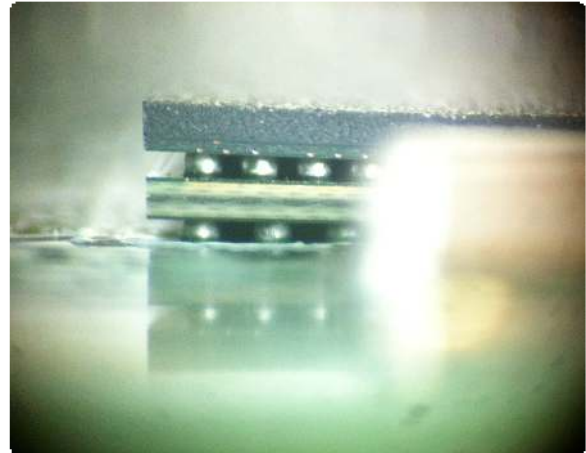
La familia Raspberry Pi

En la línea de tiempo de arriba puedes ver un conjunto de los hitos más significativos relacionados con Raspberry Pi. En la corta historia de la *Raspberry Pi Foundation* ya se han generado un respetable número de modelos:

- *Raspberry Pi modelo B* fue el primer modelo puesto a la venta, el 29 de febrero de 2012. El diseño original incluía dos modelos con el mismo circuito impreso. Ambos se diseñaron alrededor de un SoC (*System On a Chip*) de Broadcom, el [BCM2835](#) a 700MHz, pensado para aplicaciones móviles que requieran tratamiento de video o gráficos 3D (cámaras de vídeo, reproductores multimedia, teléfonos móviles, etc.). La mayoría de los pines de entrada/salida del

BCM2835 se dispusieron en una cabecera de 26 pines. El modelo B era la versión de gama alta, orientada a desarrolladores, con más memoria (512MB de RAM frente a 256MB) , un puerto Ethernet 100BaseTX y un *hub USB* con dos puertos USB 2.0. Los modelos originales incorporaban una ranura SD. En septiembre de 2012 se revisó ligeramente el diseño para corregir algunos problemas. Esta revisión modifica ligeramente los pines de entrada/salida disponibles.

- *Raspberry Pi modelo A* es la versión reducida del modelo B. Incorpora la mitad de memoria RAM que el modelo B (256MB), no incluye interfaz Ethernet y solo incorpora un puerto USB 2.0. Está pensada para aplicaciones finales donde el consumo y/o el coste sean factores importantes.



Vista lateral ampliada del montaje del BCM2835 y la memoria de los modelos originales de Raspberry Pi. Foto tomada del blog de raspberrypi.org.

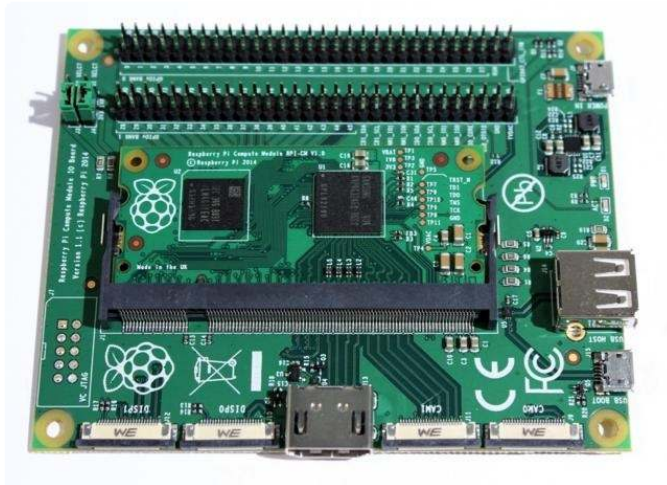
La logomarca de Raspberry Pi es una marca registrada de la *Raspberry Pi Foundation*. Fue diseñada por Paul Bleech, que ganó la competición de logos que organizó la fundación en 2011.

La mayoría de los candidatos siguen estando disponibles en el [foro](#) de la fundación. Cuidado con la logomarca, tiene unas [normas de uso](#) bastante estrictas. No puedes usarla donde te parezca.



- En abril de 2014 se anuncia la *Raspberry Pi Compute Module*. Es similar al modelo B, pero en lugar de ranura SD incorpora 4GB de eMMC Flash e integra todo en un circuito impreso DDR2 SODIMM, similar al de las memorias de los portátiles. Esto permite disponer de todos los pines del BCM2835 pero exige otro circuito impreso con el zócalo SODIMM y los conectores necesarios. Se utiliza extensivamente en el desarrollo de productos basados en Raspberry Pi, como el *media player Slice* de FiveNinjas, los satélites *CubeSat*, la cámara *Otto*, el *Cube Solver* que es capaz de solucionar un *cubo Rubik*, *Sphinx* que sirve para utilizar una tableta como un ordenador de escritorio, etc.

- En julio de 2014 se anuncia la *Raspberry Pi modelo B+*. Se trata de un rediseño del modelo B, muy similar pero con importantes consecuencias. Se amplía el número de pines de la cabecera de GPIO a 40 pines, se sustituye el zócalo SD por uno microSD, se aumenta el número de puertos USB a cuatro, se mejora la alimentación y el audio, se incluye en el mismo conector la salida de video compuesto y el audio (como muchos portátiles), y se corrige el factor de forma para que encaje completamente en el tamaño de una tarjeta de crédito. La ampliación de la cabecera de pines se complementa muy poco después con la especificación *HAT* (*Hardware Attached on Top*) que determina las limitaciones físicas y eléctricas que deben cumplir las placas de expansión de *Raspberry Pi* para garantizar la compatibilidad futura. Todas las Raspberry Pi serán compatibles con esta especificación desde esta fecha. *HAT* permite configuración automática de entradas/salidas digitales así como de los drivers por medio de dos pines dedicados (*ID_SD* e *ID_SC*). Hoy en día hay multitud de *HATs* en el mercado (como ejemplo, véase la colección de [Adafruit](#), de [Pimoroni](#) y de [The Pi Hut](#)). Además del BCM2835 el modelo B+ incorpora un SMSC LAN9514 que incorpora la interfaz Ethernet y el *hub USB* ocupando un único puerto USB del BCM2835.



Compute Module IO Board con el módulo insertado. Fuente: blog de [raspberrypi.org](#).

- En noviembre de 2014 se anuncia la *Raspberry Pi A+* como el rediseño equivalente del modelo A, sin SMSC LAN9514. Un cambio significativo es que se abandona la idea de usar el mismo circuito impreso. De esta forma se consigue reducir sensiblemente el tamaño y con ello el precio. Incorpora la cabecera HAT, que ya estará presente en



Plantilla de la especificación mecánica de HAT. Fuente: [raspberrypi.org](#).

todos los modelos posteriores, y se reduce significativamente el consumo.

- En febrero de 2015, casi en el tercer aniversario de la Raspberry Pi original, se anuncia la *Raspberry Pi 2 modelo B*. Actualiza el procesador a un [BCM2836](#) (quad-core Cortex-A7) a 900MHz e incorpora 1GB de RAM. Por primera vez se abre la posibilidad de usar Microsoft Windows 10 en *Raspberry Pi* aunque el sistema operativo recomendado por la *Raspberry Pi Foundation* sigue siendo Raspbian.
- En noviembre de 2015 se anuncia la *Raspberry Pi Zero*. Se trata de una versión diminuta del modelo A+ con mayor velocidad (1GHz) y más memoria (512MB). Sin poblar los zócalos se llegó a bajar el precio a 5\$. El número 40 de la revista [The MagPi](#) incluía una Raspberry Pi Zero de regalo. Parece que la presión de Google tras una reunión de Eric Schmidt y Eben Upton fue [decisiva](#) en su desarrollo. Sin embargo se trata de un modelo producido directamente por Raspberry Pi Trading, que tiene una capacidad de producción relativamente limitada, por lo que resulta difícil de conseguir.
- En febrero de 2016 se anuncia la *Raspberry Pi 3 modelo B*, coincidiendo con el cuarto cumpleaños de la *Raspberry Pi modelo B* original. Se vuelve a actualizar el procesador por un BCM2837 diseñado específicamente para la Raspberry Pi 3 (quad-core Cortex-A53) a 1.2GHz, incluye un BCM WiFi 802.11n, Bluetooth LE y Bluetooth 4.1 Classic. El cambio es muy importante, porque se pasa a una arquitectura de 64 bits, aunque la compatibilidad hacia atrás sigue siendo total. Además del BCM2837 incorpora un BCM43438 que implementa todas las nuevas capacidades de comunicación inalámbrica. Tan solo se deja sin conectar el receptor de radio FM.
- En febrero de 2016 Eben Upton ha anunciado en diversos foros que se está preparando un *Compute Module 3* y habrá una *Raspberry Pi 3 modelo A*. Las especificaciones no son públicas aún, pero ya hay versiones beta disponibles para [algunos ingenieros](#).

Si quieres conocer más sobre la historia de Raspberry Pi y su comunidad te recomendamos que visites el sitio web de la revista [The MagPi](#). Se trata de una revista de gran calidad y completamente gratuita en su versión electrónica. Además es una fuente muy interesante de proveedores de productos relacionados con Raspberry Pi.

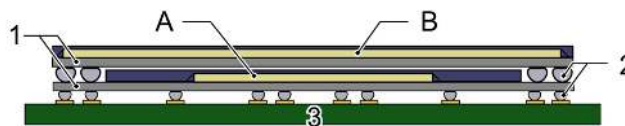
**The
MagPi**

	CM	A+	B+	2B	Zero	3B
SoC (BCMxxxx)	2835	2835	2835	2836	2835	2837
Frecuencia	700MHz	700MHz	700MHz	900MHz	1GHz	1.2GHz
Arquitectura	ARM6	ARM6	ARM6	ARM7	ARM6	ARM8
Núcleos	1	1	1	4	1	4
Memoria RAM	512MB	256MB	512MB	1GB	512MB	1GB
Memoria Flash	4GB	microSD	microSD	microSD	microSD	microSD
GPIO pins	54	26	26	26	26	26
Puertos USB	1	1	4	4	1	4
Ethernet 10/100	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
WiFi	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Bluetooth	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
HDMI	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
DSI ports	2	1	1	1	0	1
Cam ports	2	1	1	1	0	1

Los sistemas en chip de Broadcom

La mayoría de los modelos de Raspberry Pi se basan en el Broadcom BCM2835. Este *sistema-en-chip* incorpora un núcleo ARM1176JZF-S de bajo consumo y un coprocesador multimedia (GPU) de doble núcleo VideoCore IV. La GPU implementa OpenGL-ES 2.0 y es capaz de codificar y decodificar vídeo FullHD a 30fps, a la vez que muestra gráficos FullHD a 60fps en un LCD o en un monitor HDMI. Una característica llamativa de este procesador es su montaje apilado con la memoria RAM (*package on package*). La estructura se muestra en la figura. Por este motivo el circuito impreso de la Raspberry Pi no deja ver procesador alguno. Esta técnica permite reducir considerablemente el tamaño del PCB

La mayoría de los modelos de Raspberry Pi se basan en el Broadcom BCM2835. Este *sistema-en-chip* incorpora un núcleo ARM1176JZF-S de bajo consumo y un coprocesador multimedia (GPU) de doble núcleo VideoCore IV. La GPU implementa OpenGL-ES 2.0 y es capaz de codificar y decodificar vídeo FullHD a 30fps, a la vez que muestra gráficos FullHD a 60fps en un LCD o en un monitor HDMI. Una característica llamativa de este procesador es su montaje apilado con la memoria RAM (*package on package*). La estructura se muestra en la figura. Por este motivo el circuito impreso de la Raspberry Pi no deja ver procesador alguno. Esta técnica permite reducir considerablemente el tamaño del PCB (*Printed Circuit Board*) necesario.



Esquema de soldadura vertical *'package on package'*. Los sustratos del SoC (A) y de la memoria (B) se sueldan mediante una matriz de pequeñas bolitas de soldadura (2) uno encima del otro y todo ello sobre el PCB (1). Fuente: Tosaka en [Wikimedia Commons](#)

Además del procesador y la GPU, el SoC de la Raspberry Pi incorpora un amplio conjunto de periféricos:

- Temporizador.
- Controlador de interrupciones.
- Entradas/salidas digitales de propósito genérico, GPIO (*General Purpose Input-Output*). Dispone de 54 pero no todas están disponibles en la Raspberry Pi.
- Puerto USB.
- Audio PCM a través de bus I2S (*Integrated Interchip Sound*).
- Controlador de acceso directo a memoria, DMA.
- Maestro y esclavo de bus I2C (*Inter-Integrated Circuit*).
- Maestros y esclavo de bus SPI (*Serial Peripheral Interface*).
- Módulo para generación de pulsos de anchura variable, PWM.
- Puertos serie, UART.
- Interfaz para memorias eMMC, SD, SDIO.
- Interfaz HDMI

Aplicando técnicas de ingeniería inversa se documentó una parte sustancial de la GPU en el [repositorio GitHub de Herman Hermitage](#). Es posible que este trabajo haya influido en la reciente [decisión de Broadcom](#) de liberar la documentación oficial de Videocore IV. Los usuarios avanzados pueden ya disfrutar de unas [bibliotecas de desarrollo](#) razonablemente maduras y un driver OpenGL acelerado completamente libre.



El BCM2836 de la *Raspberry Pi 2 modelo B* tiene básicamente la misma arquitectura interna que su antecesor pero incorpora un procesador Cortex-A7 de cuatro núcleos que sustituye al ARM117JZF-S del BCM2835 y no utiliza la técnica de montaje *package-on-package*. Tiene importantes consecuencias desde el punto de vista del software, puesto que este procesador implementa el repertorio de instrucciones de ARM v.7 en lugar de ARM v.6 como su antecesor.

El BCM2837 de la *Raspberry Pi 3 modelo B* es un sistema-en-chip diseñado específicamente para este modelo de Raspberry Pi. Actualiza el procesador por cuatro cores Cortex-A53 de 64 bits, pero sigue incorporando la GPU Videocore IV porque se trata de una de las pocas GPUs con documentación pública del fabricante.

El ARM Cortex-A53 tiene como nombre interno *Apollo* y suele usarse en procesadores de alto rendimiento en combinación con el ARM Cortex-A57 (*Atlas*) siguiendo la configuración *big.LITTLE* (multiprocesador heterogéneo). El A53 es la versión de bajo consumo, reducido tamaño y relativa simplicidad de la arquitectura ARMv8, mientras que A57 es la versión de alto rendimiento, elevado consumo y mayor tamaño de la misma arquitectura. Por ejemplo, el Exynos 7 Octa o el Snapdragon 810 que incorporan muchos teléfonos móviles de gama alta sigue esta configuración empleando cuatro núcleos de cada tipo. En el caso del BCM2837 se opta por incluir solo cuatro núcleos de bajo consumo, por lo que no pretende competir en alto rendimiento, sino en eficiencia energética. Básicamente el rendimiento esperable es algo mejor que un quad-core Cortex-A9 (por ejemplo, como el procesador del Apple iPad 2) pero a un coste sensiblemente inferior.

El Cortex-A53 es de 64 bits, pero mantiene compatibilidad con el software de 32 bits. Esto abre la posibilidad de utilizar un sistema operativo de 32 bits y aplicaciones de 32 bits, o bien un sistema operativo de 64 bits y aplicaciones de 32 y 64 bits indistintamente. La Raspberry Pi Foundation solo distribuye de momento un sistema operativo (Raspbian GNU/Linux) de 32 bits, y no tiene planes a corto plazo de mantener una versión de 64 bits.

Una Raspberry Pi para el taller

Este taller de iniciación podría realizarse sobre cualquier modelo de Raspberry Pi. La nueva Raspberry Pi Zero es ideal para prototipos de equipos, pero dificulta notablemente el desarrollo. Por ejemplo, conectar un teclado y un ratón exigiría un hub USB OTG, la mera actualización del sistema operativo requeriría algún tipo de conexión de red, y la conexión de dispositivos externos al puerto GPIO requiere incorporar cabeceras de pines o soldadura. Todo esto está ya en la Raspberry Pi modelo B+.

El coste agregado de todos los componentes adicionales que sería preciso incorporar a la *Raspberry Pi Zero* para desarrollar cómodamente supera ampliamente el coste de la *Raspberry Pi B+*.

Por desgracia el precio del modelo B+ está subiendo debido a la caída de la demanda desde la salida al mercado de la Raspberry Pi 2 modelo B. En esta coyuntura sale al mercado la Raspberry Pi 3 modelo B al mismo precio que la Raspberry Pi 2 modelo B, pero incluyendo además interfaz WiFi y Bluetooth. Por este motivo nos decidimos por incluir la nueva Raspberry 3 modelo B en esta edición. Sin embargo todos los ejemplos del taller son compatibles con cualquiera de los modelos.

Interoperabilidad con otros productos

Alrededor de la Raspberry Pi Foundation ha surgido una enorme comunidad de usuarios de todos los niveles que genera información y productos. Hoy en día hay periféricos específicos de todo tipo para Raspberry Pi. Hay cámaras, touch panels con pantalla TFT, y multitud de tarjetas de interfaz con otros dispositivos.

Merece la pena destacar los esfuerzos por integrar periféricos de otras plataformas que ya disfrutaban de una amplia comunidad de usuarios. Así por ejemplo, pueden utilizarse los periféricos de [Lego Mindstorms](#) y las piezas de [LEGO Technic](#) para construir robots controlados por Raspberry Pi empleando [BrickPi de Dexter Industries](#).

También pueden utilizarse la enorme variedad de módulos de expansión de Arduino (*shields*) con [GertDuino de Gert van Loo](#), o con [ArduBerry de Dexter Industries](#), o con [AlaMode de WyoLum](#) o con [ArduPi de Cooking Hacks](#). El tandem de Raspberry Pi y Arduino es muy interesante cuando se requiere estricto control de la latencia. El sistema operativo *Raspbian GNU/Linux* no es de tiempo real, no puede garantizar latencias de interrupción muy bajas y la resolución de los temporizadores es del orden de los milisegundos. Sin embargo la mayor simplicidad del software de Arduino hace que sea muy predecible en cuanto a tiempos de respuesta.

Por último nos gustaría mencionar las interfaces de Raspberry Pi a [Grove](#), una arquitectura modular y abierta para construir sistemas electrónicos al estilo de LEGO en los sistemas físicos. Grove fue inicialmente concebida para ser compatible con Arduino, y por tanto pueden emplearse junto con los adaptadores mencionados arriba, pero también pueden emplearse directamente mediante [GrovePi de Dexter Industries](#), una *base shield* especialmente concebida para Raspberry Pi.

El sistema GNU/Linux

La Raspberry Pi cuenta con un completo sistema operativo, con entorno gráfico y herramientas de programación de diverso tipo. Vamos a utilizar este entorno para realizar la mayor parte del taller. Sin embargo debemos precisar que la forma de trabajo habitual en desarrollo de sistemas empotrados es que se utilice un PC y programemos la Raspberry Pi remotamente.

GNU/Linux es el nombre habitual del sistema operativo que lleva la Raspberry Pi. *Raspbian* y *Debian* no son más que distribuciones de este sistema operativo. Es decir, *Raspbian* es una selección de paquetes de GNU/Linux, compilados para una arquitectura concreta, y empaquetados con ayuda de herramientas específicas para conseguir una experiencia de usuario agradable. En lugar de ir aquí y allá en busca de instaladores y drivers como hacemos en Microsoft Windows, en GNU se especializan en conjuntos de paquetes con fines específicos.



Entorno gráfico en el que arranca el sistema inicialmente.

GNU es el nombre correcto del sistema operativo. Quiere decir *GNU's Not Unix*, es decir, GNU no es Unix. Es un acrónimo recursivo. Hace referencia a que no contiene ni una sola línea de Unix, el sistema operativo privativo de AT&T, que luego vendió a SCO y licenció a IBM, Sun, HP, Silicon Graphics, Fujitsu, Microsoft, etc. El sufijo *Linux* se refiere al kernel (núcleo) del sistema operativo. *GNU* tiene su propio kernel, el *HURD*, pero todavía no está listo para su uso general. Por eso la mayoría de las distribuciones añaden a GNU alguno de los kernels libres que hay por ahí (Linux, FreeBSD, NetBSD, etc.)

Info



En 1983 Richard M. Stallman, que trabajaba como investigador en el AI Lab del MIT, decidió empezar el proyecto GNU con el objetivo de hacer innecesario el uso de cualquiera otro software no libre. Todavía está lejos de alcanzar su objetivo pero GNU ya se utiliza en multitud de equipos electrónicos. Puedes leer más sobre el objetivo inicial del proyecto en el [El manifiesto de GNU](#) [Citation not found].

En 1985 crea la *Free Software Foundation* con el objetivo de difundir el movimiento del software libre y de ayudar al desarrollo del sistema GNU. Escucha al propio Richard Stallman [explicando la filosofía del movimiento](#).



Nada más conectar la Raspberry Pi a la alimentación arrancará en un entorno gráfico como el que se muestra en la figura al comienzo del capítulo. En la parte superior aparecen los siguientes elementos.

-  Menú de aplicaciones.
-  Terminal de línea de órdenes.
-  Herramienta de configuración de Raspberry Pi.
-  Herramienta de administración de archivos.
-  Entorno integrado de desarrollo en Python (IDLE).

-  Editor de textos.
-  Navegador básico de web.

Desde el menú es posible ejecutar la mayoría de las aplicaciones instaladas. No obstante con los botones de lanzamiento rápido de aplicaciones tendremos suficiente para la mayoría de las actividades del taller.

El sistema de archivos

Vamos primero a familiarizarnos con la estructura de carpetas y archivos del sistema. Para ello pincha sobre el botón de lanzamiento rápido del administrador de archivos. La situación será similar a la figura adjunta.

La caja de texto en la parte superior indica `/home/pi` que es la carpeta actual. Las rutas de los archivos y las carpetas utilizan el carácter `/` como separador. No es posible tener una carpeta con ese carácter en el nombre porque el sistema no podría diferenciarlo de una ruta de dos componentes. La carpeta `/` sin más es la carpeta *raíz*, de donde cuelga todo. Aquí no hay nombres de unidades, todas las unidades se ven en algún punto del árbol de carpetas que nace en la carpeta raíz. La ruta `/home/pi` hace referencia a que se encuentra en la carpeta `pi` de la carpeta

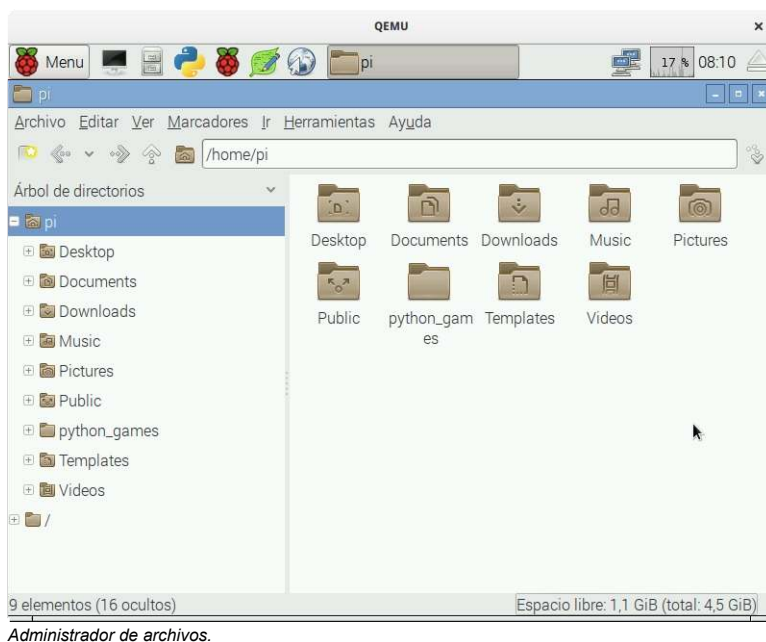
`home`. Como puedes imaginar se trata de la carpeta personal. El nombre `home` se refiere a que contiene todas las carpetas personales (casa en inglés). Y dentro de esa carpeta, la carpeta `pi` es la del usuario `pi`. Efectivamente, `pi` es el nombre del usuario creado por defecto en el sistema cuando se instala. En el taller usaremos este usuario en exclusiva, pero te animamos a que te hagas su propio usuario. Verás que en esta carpeta ya hay algunos archivos. Son ejemplos de programas en varios lenguajes de programación, que usaremos en el curso.

Aunque el sistema no lo requiere, las distintas variantes de GNU tienden a mantener una estructura común de carpetas. Por ejemplo, los siguientes suelen estar presentes en la práctica totalidad de los sistemas GNU:

- `/home/` Carpetas personales de los usuarios.
- `/root/` Carpeta personal del administrador (usuario `root`).
- `/etc/` Archivos de configuración del sistema.
- `/boot/` Archivos necesarios para el arranque del sistema.
- `/bin/` Órdenes básicas (ejecutables del sistema).
- `/usr/bin/` Resto de órdenes del sistema (ejecutables).
- `/lib/` Bibliotecas básicas del sistema (biblioteca en inglés es *library*).
- `/usr/lib/` Resto de bibliotecas del sistema.
- `/usr/local/` Software instalado de forma manual, no perteneciente al sistema.
- `/tmp/` Carpeta temporal.
- `/dev/` Dispositivos del sistema. En GNU todos los dispositivos se ven como archivos especiales.

Usa el administrador de archivos para navegar por el sistema y familiarizarte con él. No te preocupes, como usuario `pi` no puedes destruir nada esencial para el sistema. Te proponemos los siguientes ejercicios:

1. Encuentra el archivo `wpa_supplicant.conf`. Se trata del archivo donde podrás *configurar* la red WiFi para que la Raspberry Pi se conecte automáticamente a tu punto de acceso.
2. Encuentra el archivo `parpadeo.py` que es un programa de ejemplo escrito en Python que usaremos en el curso.



Administrador de archivos.

3. Encuentra el programa `gcc` . Se trata del compilador de C.
4. Encuentra el programa `idle` . Se trata del entorno integrado de programación con Python.

Warning

Tradicionalmente en sistemas operativos se utiliza el término *directorio* para referirse a una carpeta. Del mismo modo muchos textos en español hablan de *ficheros* para referirse a archivos. Nosotros intentaremos utilizar el término *carpeta* (*folder* en inglés) que encaja mejor en la metáfora del escritorio.

File es archivo en inglés. Un *file* es una de esas carpetas de cartón que se meten en los archivadores de oficina. El problema es que no se puede llamar carpeta también a los archivos. Por eso se buscaron traducciones más neutras. Fichero es realmente el archivador, más que el contenido del archivador. Así que *archivo* nos parece una traducción más correcta.

Pero te avisamos porque en la documentación que leas por ahí es fácil que aparezcan. **Directorio es lo mismo que carpeta y fichero es lo mismo que archivo.**

El entorno de línea de órdenes



Ejecuta la terminal de línea de órdenes pulsando sobre el icono correspondiente. Aunque aparentemente se trata de una interfaz primitiva ésta es una de las formas más flexibles para comunicarse con el sistema operativo.

Al pulsar el icono veremos que se abre una nueva ventana. Esa ventana corresponde al programa simulador de terminal. Se comporta como una consola antigua con teclado y pantalla alfanumérica. A su vez el programa terminal ejecuta otro programa que se encarga de interpretar las órdenes textuales, la *shell*. La *shell* es el intermediario entre el usuario y el sistema operativo.

En GNU/Linux la *shell* que se utiliza normalmente se llama `bash` (*Bourne Again SHell*). Tiene multitud de características que la convierten en un completo lenguaje de programación por sí misma. Nosotros no veremos las características avanzadas, sino unas nociones básicas que te permitirán desenvolverte con soltura durante el curso.

Cuando se ejecuta la *shell* aparece un pequeño texto antes del cursor, es el *prompt*.

```
pi@raspberrypi:~ $ _
```

Antes de los dos puntos aparece el usuario y el nombre del ordenador simulando una dirección de correo electrónico. Antes del símbolo `@` aparece el nombre del usuario que ejecuta la *shell*. En este caso el usuario es `pi`, que es el usuario por defecto, y el que usaremos en nuestros ejemplos. Después aparece el nombre del *host*, que hemos configurado en la instalación como `rpi`.

Después de los dos puntos y antes del símbolo `$` aparece la carpeta de trabajo. La carpeta (o el directorio) de trabajo es aquella carpeta en la que se encuentra actualmente la *shell*. Todos los procesos tienen una carpeta de trabajo y la *shell* no es una excepción. Se utiliza como base para determinar los archivos que se localizan mediante *rutras relativas*. Veremos esto enseguida.

El símbolo `~` es una abreviatura para la carpeta *home* del usuario. En este caso `/home/pi`. Puede utilizarse esta abreviatura en cualquier orden que necesite una ruta.

Hay muchas referencias muy recomendables para entender la *shell* y explotar todo su potencial. Un excelente libro disponible de forma gratuita es [Citation not found] disponible en tldp.org. Otra referencia actual y muy completa es [Citation not found].

Gestión de archivos

Todas las operaciones que pueden realizarse con el administrador de archivos también pueden realizarse con órdenes en la *shell*. Veamos un breve resumen de las órdenes más frecuentes.

Listado de archivos: `ls`

La operación más básica de gestión de archivos es mostrar el contenido de una carpeta. Se realiza con la orden `ls` (*list*). Sin más argumentos muestra el contenido de la carpeta de trabajo (la que aparece en el *prompt*). Por ejemplo:


```
pi@raspberrypi:~ $ ls
Desktop  Documents  Music      Public      src          Videos
doc      Downloads  Pictures   python_games  Templates
pi@raspberrypi:~ $
```

Los colores de cada elemento nos indica de qué se trata. En azul se muestran las carpetas, en gris los archivos normales, y en verde los ejecutables.

Por defecto, no muestra los elementos ocultos, que son aquéllos cuyo nombre comienza con `.` (punto). Se puede indicar como argumento la carpeta o las carpetas cuyo contenido se quiere listar:

```
pi@raspberrypi:~ $ ls src
c  python  README.md
pi@raspberrypi:~ $
```

Se asume que se quiere listar la carpeta `src` dentro de la carpeta actual. Es lo que se conoce como una *ruta relativa* y se asume que es relativa a la carpeta de trabajo. También se puede indicar la ruta completa `/home/pi/src` que se conoce como *ruta absoluta*. Como dijimos anteriormente el símbolo `~` es una abreviatura de la carpeta *home* del usuario. Por tanto otra forma de expresar la ruta completa sería `~/src`.

Y para ver los archivos y carpetas ocultos se puede usar la opción `-a`:

```
pi@raspberrypi:~ $ ls src -a
.  ..  c  .git  python  README.md
pi@raspberrypi:~ $
```

En este caso la carpeta `.git` es oculta. Las carpetas `.` y `..` tienen un significado especial para el sistema operativo y aparecen en todas las carpetas del sistema. La carpeta `.` representa la misma carpeta en la que se encuentra, y la carpeta `..` representa la carpeta que contiene a la que se muestra. En nuestro ejemplo la carpeta `src/..` es `/home/pi` mientras que la carpeta `..` sería `/home`. La carpeta `src/.` es la misma que `./src`, la misma que `src` y la misma que `/home/pi/src` en nuestro ejemplo.

Como el resto de órdenes `ls` permite especificar varias opciones que modifican su comportamiento. Estas opciones vienen precedidas por un guión y pueden combinarse o indicarse por separado. Por ejemplo:

```
pi@raspberrypi:~ $ ls src -a -F
./  ../  c/  .git/  python/  README.md
pi@raspberrypi:~ $ ls -aF src
./  ../  c/  .git/  python/  README.md
pi@raspberrypi:~ $
```

Ambas formas son equivalentes. La opción `-F` añade un carácter al final de cada nombre para expresar el tipo de elemento del que se trata (`/` si es una carpeta, `*` si es un ejecutable o nada si es un archivo normal). Ya no se suele usar esta opción porque los colores son una representación más intuitiva.

Una opción interesante es `-l` (*long*) que muestra metainformación sobre permisos, usuarios, tamaño y fecha de los archivos. Por ejemplo, supongamos que queremos ver la fecha del archivo `src/README.md`.

```
pi@raspberrypi:~ $ ls src/README.md -l
-rw-r--r-- 1 pi pi 55 Apr 11 08:26 src/README.md
pi@raspberrypi:~ $
```

Ya veremos qué significa todo esto. De momento es suficiente con entender que la fecha es lo que aparece antes del nombre (`Apr 11 08:26`, 11 de abril a las 8:26 de la mañana).

En la siguiente tabla se muestra un conjunto de opciones frecuentemente usadas.

Opción	Significado
-l	Formato de listado largo, con información de permisos, usuarios, tamaños, fecha, etc.
-R	Lista las subcarpetas de manera recursiva
-1	No agrupa los resultados, muestra cada archivo en una línea
-d	No muestra el contenido de la carpeta, sino el nombre de la carpeta
-a	Muestra también los archivos ocultos, aquellos que empiezan por <code>.</code>
--help	Muestra la ayuda de la orden, junto con una lista extendida de las opciones del mismo

La *shell* admite un conjunto de abreviaturas que se utilizan como comodines a la hora de escribir rutas de carpetas o archivos. Así, por ejemplo, el carácter `*` representa cualquier combinación de caracteres y el carácter `?` representa un carácter individual. Por ejemplo, veamos todos los archivos C de las carpetas de `src/c/reactor`.

```
pi@raspberrypi:~ $ ls src/c/reactor/*/*.c
src/c/reactor/reactor/blink_handler.c
src/c/reactor/reactor/console.c
...
```

Fíjate como usamos un `*` para representar cualquier carpeta dentro de `src/c/reactor` y luego un `*.c` para representar cualquier nombre de archivo que termina en `.c`.

Cambiar carpeta de trabajo: `cd`

La orden `cd` (*change directory*) cambia la carpeta de trabajo a la indicada. Si no se especifica ninguna ruta va a la carpeta *home* del usuario. Por ejemplo:

```
pi@raspberrypi:~ $ cd src/c/reactor
pi@raspberrypi:~/src/c/reactor $ _
```

Fíjate en cómo cambia el *prompt*. Ahora indica que la carpeta de trabajo es `~/src/c/reactor`, o lo que es lo mismo, `/home/pi/src/c/reactor`. Para subir un nivel en la jerarquía, es decir, para ir a la carpeta `/home/pi/src/c` bastaría con escribir:

```
pi@raspberrypi:~/src/c/reactor $ cd ..
pi@raspberrypi:~/src/c $ _
```

Ubicación actual: `pwd`

El *prompt* ya muestra la carpeta de trabajo pero a veces necesitamos ese texto para usarlo en otra orden. Para eso podemos usar la orden `pwd` (*print working directory*):

```
pi@raspberrypi:~/src/c $ pwd
/home/pi/src/c
pi@raspberrypi:~/src/c $ _
```

Creación de carpetas: `mkdir`

La orden `mkdir` permite la creación de nuevas carpetas en aquellos puntos en los que el usuario tiene permisos para ello. Por ejemplo:

```
pi@raspberrypi:~/src/c $ cd
pi@raspberrypi:~ $ mkdir test
pi@raspberrypi:~ $ cd test
pi@raspberrypi:~/test $ _
```

La primera orden cambia la carpeta de trabajo al *home* del usuario. Allí crea la carpeta `test` y posteriormente cambia la carpeta de trabajo a ésta.

Borrado de archivos y carpetas: `rm`

La orden `rm` (*remove*) se utiliza para borrar archivos y carpetas. No se puede deshacer, así que presta mucha atención cuando la uses. Por ejemplo:

```
pi@raspberrypi:~/src/c $ cd
pi@raspberrypi:~ $ rm -r test
pi@raspberrypi:~ $ ls test
ls: cannot access test: No such file or directory
pi@raspberrypi:~ $
```

Entre opciones que acepta esta orden, destacamos las siguientes:

Opción	Significado
<code>-r</code>	Procesa subcarpetas de forma recursiva
<code>-i</code>	Pide confirmación para cada borrado
<code>-f</code>	Forzado, ignora archivos no existentes y elimina cualquier aviso de confirmación

Para borrar carpetas existe una orden `rmdir` específica, que solo borra una carpeta si está vacía. Es más habitual usar `rm` porque también borra el contenido de la carpeta si no está vacía. El problema es que es mucho más peligrosa. No olvides hacer copias de seguridad.

Copia de archivos y carpetas: `cp`

La orden `cp` (*copy*) se usa para copiar tanto archivos como carpetas. Veamos un ejemplo:

```
pi@raspberrypi:~ $ cp -r src/c/reactor/test .
pi@raspberrypi:~ $ ls test
makefile          test_delayed_handler.c  test_pipe_handler.c
test_acceptor.c   test_Event_handler.c    test_process_handler.c
...
pi@raspberrypi:~ $
```

La opción `-r` indica que queremos una copia *recursiva*. Si hay carpetas copiará el contenido de las carpetas. La primera ruta es el *origen* (`src/c/reactor/test`) y la segunda ruta es el *destino* (`.`). Por tanto este ejemplo copia la carpeta `test` que contiene la carpeta `src/c/reactor` en la carpeta actual.

Renombrar archivos y carpetas: `mv`

La orden `mv` (*move*) se utiliza como la orden `cp`, con la diferencia de que en lugar de crear una copia del objeto, la orden `mv`, mueve el objeto *origen* a un *destino*. Por supuesto se puede usar simplemente para cambiar el nombre a carpetas o archivos:

```
pi@raspberrypi:~ $ mv test test-reactor
pi@raspberrypi:~ $ ls test
ls: cannot access test: No such file or directory
pi@raspberrypi:~ $
```

Mostrar el contenido: `cat`

La última operación de gestión de archivos que vamos a mencionar por el momento es la orden `cat` (*concatenate*), que muestra por pantalla el contenido de uno o más archivos.

```
pi@raspberrypi:~ $ cat src/c/reactor/AUTHORS
Contribuidores más importantes de Reactor:

* Francisco Moya <francisco.moya@uclm.es>
pi@raspberrypi:~ $
```

Documentación y ayuda

Cualquier sistema complejo necesita tiempo para dominarse y GNU/Linux no es una excepción. Si no has usado GNU hasta ahora te quedan muchas cosas por aprender y tendrás que tener paciencia. Pero la buena noticia es que todos los sistemas complejos incorporan un sistema de ayuda para facilitar este proceso de aprendizaje. Aprende a tu ritmo pero sobre todo quédate con la idea más importante, cómo conseguir ayuda en GNU.

La mayoría de las órdenes tienen su propia ayuda en línea con la opción `--help` o en algunos casos más primitivos `-h`. Si tienes dudas no dudes en usar esta opción. Lo máximo que puede ocurrir es que el programa te indique no conoce esa opción. Por ejemplo:

```
pi@raspberrypi:~ $ gpio --help
gpio: Unknown command --help.
pi@raspberrypi:~ $ gpio -h
gpio: Usage: gpio -v
        gpio -h
        ...
pi@raspberrypi:~ $
```

Si esta opción no ha resuelto tus dudas usa `man` (*manual*). Se trata de una herramienta disponible en todas las variantes de Unix para consultar la versión electrónica del manual de referencia. ¿Y si tienes dudas de cómo usar `man`? No lo pienses, usa `man --help` y si no es suficiente `man man`. Recuerda que para salir de `man` hay que pulsar la tecla `q` (*quit*).

Hemos instalado las páginas de manual en castellano. Si existe ayuda en castellano la verás en castellano. Si no, practica el inglés, si usas la Raspberry Pi te será muy útil saber inglés.

En algunos casos hay varias páginas de manual para el mismo concepto. Por ejemplo, `printf` es una orden de la *shell* pero también es una función C y probablemente tienes la versión en castellano y en inglés. Las páginas de manual se agrupan en secciones, las secciones originales del manual de referencia de Unix. Se puede indicar a `man` a qué sección queremos referirnos (un número de 1 a 9). Por ejemplo:

```
pi@raspberrypi:~ $ man 3 printf
```

La lista completa de secciones la tienes en la página de manual de `man`, no hace falta que te la sepas de memoria.

Algunas opciones de `man` que merece la pena destacar:

Opción	Significado
<code>-k</code>	Busca la palabra clave indicada en las páginas de manual y muestra las que la contienen
<code>-a</code>	Muestra todas las páginas de manual que correspondan

Pipes y redirecciones

La salida estándar de un programa puede ser demasiado grande. Tan grande que ni siquiera con la barra de desplazamiento del terminal podrías ver todo. En esos casos puede que te interese *redirigir* la salida a un archivo para su lectura detenida con un editor. Por ejemplo, vamos a generar un archivo con todos los ejecutables del sistema:

```
pi@raspberrypi:~ $ ls -l /bin > exe-bin.txt
pi@raspberrypi:~ $ ls -l /usr/bin > exe-usr-bin.txt
pi@raspberrypi:~ $ ls -l /usr/local/bin > exe-usr-local-bin.txt
pi@raspberrypi:~ $
```

Al utilizar el símbolo `>` le estamos diciendo a la *shell* que cuando tenga que escribir algo por salida estándar lo escriba en el archivo indicado. En nuestro ejemplo cuando el programa `ls` ejecuta una llamada a *printf* o similar lo que escribe va directamente al final del archivo indicado. Ahora podemos ver estos archivos con detenimiento usando un editor, como *leafpad*:

```
pi@raspberrypi:~ $ leafpad exe-bin.txt
```

¿Y si queremos contar cuántos ejecutables tiene en total el sistema? Podríamos usar el programa `wc` que cuenta entre otras cosas las líneas de un archivo. Por ejemplo:

```
pi@raspberrypi:~ $ wc -l exe-*.txt
 158 exe-bin.txt
1168 exe-usr-bin.txt
   1 exe-usr-local-bin.txt
1327 total
pi@raspberrypi:~ $
```

También podemos redirigir la entrada estándar de un programa. Por ejemplo:

```
pi@raspberrypi:~ $ wc -l < exe-bin.txt
158
pi@raspberrypi:~ $ wc -l < exe-usr-bin.txt
1168
pi@raspberrypi:~ $ wc -l < exe-usr-local-bin.txt
1
pi@raspberrypi:~ $
```

Ahora `wc` no indica ningún nombre de archivo porque no lo conoce, le llega directamente el contenido del archivo cuando llama a *scanf* o similar. Lo bueno es que el resultado es un simple número que puede ser más conveniente para otras cosas.

Pero si estamos interesados en saber cuántos ejecutables hay y no nos interesa qué archivos concretos son ¿para qué guardamos la lista de ejecutables en archivos? En GNU es posible conectar la salida estándar de un programa con la entrada estándar de otro programa estableciendo lo que se conoce como una *tubería* (*pipe* en inglés):

```
pi@raspberrypi:~ $ ls -l /bin /usr/bin /usr/local/bin | wc -l
1327
pi@raspberrypi:~ $
```

El símbolo `|` conecta la salida estándar de la orden de la izquierda con la entrada estándar de la orden de la derecha.

Incluso si queremos examinar la lista de ejecutables podemos usar esta construcción para evitar tener que guardar el archivo intermedio:

```
pi@raspberrypi:~ $ ls -l /bin /usr/bin /usr/local/bin | less
```

El programa `less` es un *paginador*, un programa que va mostrando lo que le llega por entrada estándar página a página. Con los cursores o con el espacio podemos recorrer toda la información y cuando hayamos terminado de examinarla basta salir presionando la tecla `q`.

Pero los programas no solo tienen una salida estándar. También tienen una salida de error estándar. Cuando muestran un error suelen mostrarlo por esta *salida de error*. Normalmente es el mismo terminal y vemos los mensajes de error mezclados con la salida del programa. Pero podemos separarla:

```
pi@raspberrypi:~ $ ls -R /etc 2>errores.txt
/etc/:
adduser.conf
...
pi@raspberrypi:~ $
```

El número 2 hace referencia al *descriptor de archivo* correspondiente a la salida de error estándar, mientras que el 1 (o nada) hace referencia al *descriptor de archivo* de la salida estándar. Hablaremos de descriptores de archivo cuando lleguemos a la parte de programación del taller.

Si solo estamos interesados en los errores podemos descartar la salida estándar redirigiéndola al archivo especial

```
/dev/null :
```

```
pi@raspberrypi:~ $ ls -R /etc >/dev/null
ls: no se puede abrir el directorio /etc/polkit-1/localauthority: Permiso denegado
ls: no se puede abrir el directorio /etc/ssl/private: Permiso denegado
pi@raspberrypi:~ $
```

Como usuario `pi` no podemos hacer cualquier cosa. El problema de esta orden es que si hay muchos errores perdemos los primeros, que suelen ser los más importantes. ¿Podemos usar un paginador? Pues no, porque las tuberías solo conectan salida estándar con entrada estándar. La solución es conectar la salida de error estándar con la misma salida estándar, vaya a donde vaya.

```
pi@raspberrypi:~ $ ls -R /etc 2>&1 1>/dev/null | less
```

Redirige el descriptor 2 (error estándar) al descriptor 1 (salida estándar), el descriptor 1 (salida estándar) a `/dev/null` (descartar) y conecta la nueva salida estándar (nuevo descriptor 1 que es la antigua salida de error) a la entrada estándar del paginador `less`.

O bien sin descartar la salida estándar, combinándola junto con la salida de error estándar:

```
pi@raspberrypi:~ $ ls -R /etc 2>&1 | less
```

Esta última forma de redirección es extraordinariamente útil cuando se programa en lenguaje C. Cuando se compilan programas aparecen mensajes de la herramienta de compilación y también errores del compilador. Suele ser mucho texto, necesitamos verlo con un paginador y nos interesa especialmente los primeros errores.

Además de los paginadores conviene comentar en esta sección algunos *filtros*. Un filtro no es más que un programa que recibe la entrada estándar de otro programa, la manipula de forma determinada, y la vuelve a sacar una vez manipulada por la salida estándar.

Un filtro extraordinariamente útil es `grep` (*global regular expression print*). Se trata de una orden que busca líneas que contienen un patrón determinado. Las líneas que no contienen el patrón no se imprimen, mientras que las que lo contienen se imprimen. Por ejemplo para encontrar todos los ejecutables de `/usr/bin` que contienen `zip` en el nombre:

```
pi@raspberrypi:~ $ ls -R /usr/bin | grep zip
pi@raspberrypi:~ $
```

Puede cambiarse el funcionamiento para que imprima las líneas que no cumplen el patrón, o escribir patrones mucho más evolucionados. Cuando puedas tómate algo de tiempo para entender el funcionamiento de `grep` leyendo la página de manual.

Enlaces: 1n

Un archivo tiene dos partes claramente distintas: un nombre y un contenido. El nombre permite organizar y encontrar los archivos fácilmente, pero una vez encontrado un programa puede *abrirlo* (por ejemplo, con la función *open* si estamos programando en C) y olvidarse completamente del nombre.

La idea de que el nombre es un mero artefacto organizativo nos lleva a preguntarnos si podríamos usar varios nombres para un mismo archivo. La respuesta es afirmativa, con un *enlace*. La forma más simple de enlace convierte en indistinguible el archivo origen del archivo destino. Examina en detalle esta secuencia de órdenes:

```
pi@raspberrypi:~ $ echo "Primera prueba" > a.txt
pi@raspberrypi:~ $ cat a.txt
Primera prueba
pi@raspberrypi:~ $ ln a.txt b.txt
pi@raspberrypi:~ $ echo "Segunda prueba" > b.txt
pi@raspberrypi:~ $ cat a.txt
Segunda prueba
pi@raspberrypi:~ $
```

La orden `ln a.txt b.txt` ha creado un nuevo nombre `b.txt` para el mismo archivo `a.txt`. Por eso cuando escribimos en `b.txt` el resultado puede verse también en `a.txt`.

El problema es que esta forma de enlaces no funciona con las carpetas, que son también entidades organizativas sin otro contenido que los nombres de sus archivos. Tampoco funcionaría con archivos que están en otro *sistema de archivos*, como por ejemplo en un pincho USB.

Para solucionar estos problemas están los enlaces simbólicos (opción `-s`):

```
pi@raspberrypi:~ $ ln -s src/c/reactor reactor
pi@raspberrypi:~ $ ls -l reactor
reactor -> src/c/reactor
pi@raspberrypi:~ $
```

Un enlace simbólico es simplemente una indicación de que ese nombre corresponde realmente al contenido etiquetado con otro nombre. Esto resuelve los problemas anteriores pero ya no son equivalentes el archivo y el enlace. Si se borra el archivo el enlace se queda colgando (*dangling link*) y cualquier intento de acceso produciría un error.

Los enlaces simbólicos son muy útiles para simplificar la navegación por carpetas. El último ejemplo que mostrábamos líneas arriba es ilustrativo de esto. Si estamos trabajando con una carpeta continuamente (`reactor`) es lógico dejarla más a mano en nuestro *home*.

Usuarios y permisos

Ya habrás notado que como usuario `pi` no tienes permisos para hacer cualquier cosa. Vamos a ver con más detalle el modelo de permisos de Unix y cómo puedes saltarte las restricciones cuando sea necesario.

La orden `id` permite saber quién eres:

```
pi@raspberrypi:~ $ id
uid=1000(pi) gid=1000(pi) grupos=1000(pi),4(adm),20(dialout),24(cdrom),27(sudo),
29(audio),44(video),46(plugdev),60(games),100(users),101(input),108(netdev),997(
gpio),998(i2c),999(spi)
pi@raspberrypi:~ $
```

El `uid` es el identificador de usuario, mientras que el `gid` es el identificador de grupo. Todos los usuarios tienen un `uid` y un `gid`. Sin embargo un usuario puede pertenecer a muchos grupos, como se muestra al final de la salida de `id`. Hay un usuario especial, el *super-usuario*, que tiene el `uid 0` y no tiene ninguna limitación de permisos.

Vamos a examinar ahora con más detalle esta orden `ls` que vimos anteriormente:

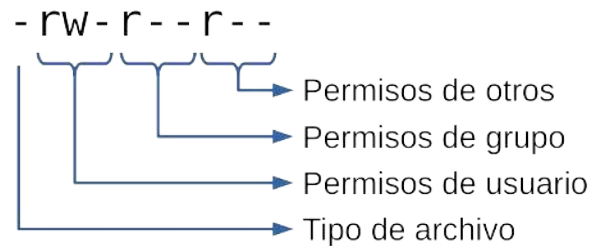
```
pi@raspberrypi:~ $ ls src/README.md -l
-rw-r--r-- 1 pi pi 55 Apr 11 08:26 src/README.md
pi@raspberrypi:~ $
```

Hasta ahora solo habíamos visto la fecha y hora de la última modificación. Sin embargo antes de la fecha hay mucha más información.

Campo	Significado
<code>-rw-r--r--</code>	Permisos
<code>1</code>	Número de referencias al archivo (nombres o enlaces)
<code>pi</code>	Usuario (dueño)
<code>pi</code>	Grupo (dueño)
<code>55</code>	Tamaño en bytes
<code>Apr 11 08:26</code>	Fecha y hora

Los permisos se representan en una notación abreviada que corresponde de forma directa con la representación interna en el sistema de archivos.

- El primer caracter representa el tipo de archivo. Es `d` para los directorios, `-` para los archivos normales, `l` para los enlaces simbólicos, u otras letras que representan otro tipo de archivos (dispositivos de varios tipos, pipes, sockets, etc.).
- Los siguientes caracteres están divididos en grupos de tres letras que representan los permisos para el usuario dueño del archivo, para el grupo dueño del archivo y para todos los demás, en este orden.



Estructura de los permisos de un archivo en GNU.

Permiso	Representación
Lectura	<code>r</code> en la primera letra
Escritura	<code>w</code> en la segunda letra
Ejecución	<code>x</code> en la tercera letra
<i>Set-Id</i>	<code>s</code> en la tercera letra (solo en grupo de usuario o grupo)
Pegajoso	<code>t</code> en la tercera letra (solo en grupo de otros)

El permiso de lectura permite ver el contenido del archivo al grupo que corresponda. En nuestro ejemplo los tres grupos (usuario dueño, grupo dueño y otros) pueden leer el archivo. Es decir, podremos ejecutar la orden `cat`, o leerlo con un editor como `leafpad`. En el caso de las carpetas el permiso de lectura permite ver el contenido, es decir, hacer `ls`.

El permiso de escritura permite hacer cambios en el archivo. En nuestro ejemplo solo el usuario dueño `pi` puede modificarlo, porque ninguno de los otros grupos de permisos tiene la letra `w`. En una carpeta significa que podemos añadir o borrar elementos (otros archivos o carpetas).

El permiso de ejecución permite ejecutar el archivo. Es ésto y ninguna otra cosa lo que convierte en ejecutable un archivo. En GNU no hay extensiones especiales para identificar los ejecutables. En las carpetas significa que podemos acceder a lo contenido en ellos. Eso implica tanto hacer un `cd` como acceder de cualquier forma al contenido de lo que hay en la carpeta (ver el tamaño o el dueño de un archivo, mostrar el contenido o listar una subcarpeta).

El permiso *set-id* sirve para fijar el *uid* o el *gid* al del dueño del archivo cuando se ejecuta. Es decir, si un ejecutable tiene el permiso *set-id* para el usuario dueño entonces cuando se ejecuta lo hace con los permisos de ese usuario dueño (cambia temporalmente su *uid*) y no con los permisos del usuario que lo ejecuta. Es una forma de ceder permisos para operaciones determinadas a otros usuarios, a todos los que tengan permiso para ejecutar el archivo. Lo mismo puede hacerse con el permiso *set-id* para el grupo dueño. Cuando se ejecuta el archivo lo hace como si el usuario que lo ha ejecutado perteneciera al grupo dueño del archivo (cambia temporalmente su *gid*). A este permiso se le llama normalmente *setuid* o *setgid* dependiendo si se aplica al usuario dueño o al grupo dueño.

El permiso pegajoso (*sticky* en inglés) ha cambiado su significado con la evolución histórica de Unix. Hoy en día es muy útil para las carpetas. Una carpeta con el permiso pegajoso no permite que los archivos contenidos sean borrados o renombrados por otros usuarios que no sean el dueño. Se usa por ejemplo en la carpeta temporal `/tmp` para que unos usuarios no puedan provocar problemas a otros.

Los permisos se cambian con la orden `chmod` (*change mode*) y el grupo dueño con `chgrp` (*change group*):

```
pi@raspberrypi:~ $ chgrp users src/README.md
-rw-r--r-- 1 pi users 55 Apr 11 08:26 src/README.md
pi@raspberrypi:~ $ chmod g+w src/README.md
-rw-rw-r-- 1 pi users 55 Apr 11 08:26 src/README.md
pi@raspberrypi:~ $
```

Hemos cambiado el grupo del archivo por `users`, que es uno de los grupos al que pertenecemos, y posteriormente hemos añadido el permiso de escritura al grupo dueño. Ahora todos los usuarios pertenecientes al grupo `users` pueden editar el archivo. La orden `chmod` permite añadir permisos (con `+`) o eliminarlos (con `-`) para el usuario dueño (con `u`), el grupo dueño (con `g`), otros (con `o`) o todos (con `a` de *all*).

Permiso <code>chown</code>	Significado
<code>u+r</code>	Añadir permiso de lectura al usuario dueño
<code>u+w</code>	Añadir permiso de escritura al usuario dueño
<code>u+x</code>	Añadir permiso de ejecución al usuario dueño
<code>u+s</code>	Añadir permiso <i>setuid</i>
<code>g+r</code>	Añadir permiso de lectura al grupo dueño
<code>g+w</code>	Añadir permiso de escritura al grupo dueño
<code>g+x</code>	Añadir permiso de ejecución al grupo dueño
<code>g+s</code>	Añadir permiso <i>setgid</i>
<code>o+r</code>	Añadir permiso de lectura para otros
<code>o+w</code>	Añadir permiso de escritura para otros
<code>o+x</code>	Añadir permiso de ejecución para otros
<code>+t</code>	Añadir permiso pegajoso

También se pueden combinar tanto el grupo al que se aplica el permiso como los permisos que se aplican. Con `-R` se aplica de forma recursiva. Por ejemplo:

```
pi@raspberrypi:~ $ chmod -R ug+rwX src
pi@raspberrypi:~ $
```

Esto aplica los permisos de lectura (`r`), escritura (`w`) y ejecución (`x`) para el usuario dueño (`u`) y el grupo dueño (`g`) a la carpeta `src` y todo su contenido. Fíjate que el permiso de ejecución lo hemos puesto con mayúscula en lugar de minúscula. Eso tiene un significado especial. Cuando a `chmod` se le indica `X` en lugar de `x` le estamos diciendo que solo aplique ese permiso a carpetas, no a los archivos.

El super-usuario

El *super-usuario* es el usuario con *uid* cero, que normalmente tiene el nombre de `root`. A este usuario no se le aplican restricciones de permisos. Puede hacerlo todo, es el administrador del sistema. Evidentemente es necesario ser administrador para poder hacer ciertas cosas, como actualizar el sistema, instalar nuevo software, etc. Pero también va a ser necesario para utilizar algunos periféricos de la Raspberry Pi. Por ejemplo, para leer o cambiar los valores de las patitas de GPIO (*General Purpose Input Output*).

En la Raspberry Pi se considera tan necesario que está configurada para que sea extremadamente simple convertirse en superusuario. Basta ejecutar `sudo su`:

```
pi@raspberrypi:~ $ sudo su
pi@raspberrypi:~ # id
uid=0(root) gid=0(root) grupos=0(root)
pi@raspberrypi:~ #
```

Fíjate cómo ha cambiado el *prompt*. Aparece una `#` que debe interpretarse como una severa advertencia. ¡Cuidado! ¡Puedes destruirlo todo!

Procura minimizar el tiempo que estás como *superusuario*. No es infrecuente destruir completamente el software del sistema por error y tendrías que instalarlo todo desde cero.

La orden `sudo` (*superuser do*) permite a determinados usuarios y con ciertas restricciones ejecutar como superusuario cualquier orden del sistema. Como puedes imaginar es un programa que tiene el permiso *setuid* de `root`. No hace falta ejecutar `sudo su`, se puede ejecutar `sudo` seguido de cualquier cosa que tengamos que hacer como superusuario. Por ejemplo:

```
pi@raspberrypi:~ $ sudo chown root ejecutable
pi@raspberrypi:~ $ sudo chmod u+s ejecutable
pi@raspberrypi:~ $
```

Hemos cambiado el dueño de un ejecutable usando la orden `chown`. Como superusuario podemos cambiar cualquier cosa. Hemos puesto como nuevo dueño a `root` (el superusuario) y hemos añadido el permiso *setuid*. A partir de este momento `ejecutable` se ejecuta con permisos de superusuario.

Gestión de procesos

Una de las actividades que seguramente deberás hacer cuando estás desarrollando es ver qué procesos están ejecutándose en el sistema y parar procesos que puedan haberse quedado bloqueados.

Vamos a comentar solo tres aplicaciones para estos fines aunque la gama de herramientas es mucho más amplia. No te quedes en lo que te contamos, aprende poco a poco más herramientas.

En primer lugar para ver los procesos que se ejecutan en el sistema está la orden `ps` (*processes*). Sin ningún argumento adicional nos proporciona información de los procesos que se han ejecutado desde la misma *shell* en la que se ejecuta `ps`:

```
pi@raspberrypi:~ $ ps
  PID TTY          TIME CMD
   894 pts/0    00:00:18 bash
  1211 pts/0    00:00:00 ps
pi@raspberrypi:~ $
```

En un formato tabular se muestra información básica:

Campo	Significado
PID	Identificador de proceso
TTY	Terminal
TIME	Tiempo acumulado de CPU
CMD	Nombre del ejecutable

El *pid* (*process id*) es un número que asigna el sistema operativo a todos los procesos. Cada proceso en ejecución debe tener un *pid* diferente, pero una vez terminado se puede emplear su antiguo *pid* en otro proceso. Podemos controlar los procesos usando este número de identificación.

Vamos a ilustrarlo con un ejemplo. Abre un terminal y escribe:

```
pi@raspberrypi:~ $ cat
```

El proceso `cat` está esperando datos de su entrada estándar, pero vamos a suponer que no tenemos ni idea de qué pasa. Solo sabemos que el programa no termina. Vamos a matarlo.

Ejecuta otro terminal y escribe:

```
pi@raspberrypi:~ $ ps -u pi | grep cat
 2093 pts/0    00:00:00 cat
pi@raspberrypi:~ $
```

Mostramos todos los procesos del usuario `pi` (opción `-u`) pero luego filtramos la salida con `grep` para mostrar solo las líneas que contienen `cat`. El primer número es el PID del proceso. Vamos a solicitar su terminación con la orden `kill`:

```
pi@raspberrypi:~ $ kill 2093
pi@raspberrypi:~ $
```

Si te fijas en el primer terminal el proceso ha terminado y lo indica en pantalla con un mensaje.

```
pi@raspberrypi:~ $ cat
Terminado
pi@raspberrypi:~ $
```

Es posible que el programa no termine. Aún podemos hacer algo más, podemos exigir que el programa termine. Si esta vez no lo hace el sistema operativo lo matará. Vuelve a ejecutar `cat` y en la otra ventana:

```
pi@raspberrypi:~ $ ps -u pi | grep cat
2101 pts/0    00:00:00 cat
pi@raspberrypi:~ $ kill -9 2101
pi@raspberrypi:~ $
```

Ahora el mensaje que muestra la otra ventana es ligeramente diferente:

```
pi@raspberrypi:~ $ cat
Terminado (killed)
pi@raspberrypi:~ $
```

La orden `kill` no es exactamente para *matar* procesos. En realidad `kill` envía señales a los procesos. Hay un montón de señales que se pueden mandar (usa la opción `-l` para tener una lista). Por defecto envía la señal 15 (`SIGTERM`) que es una solicitud de terminación. Los procesos pueden ignorarla. La opción `-9` simplemente manda una señal diferente (`SIGKILL`) que los procesos no pueden ignorar.

También conviene comentar la orden `top`, que muestra el estado del sistema en tiempo real, poniendo los procesos que más memoria y/o CPU consumen en los primeros lugares. Si el sistema está lento y no sabes por qué puede que hayas dejado algún proceso colgando. Mira con `top` cuál es y mátao. Una vez identificado puedes matarlo desde el propio `top` pulsando la tecla `k` (*kill*). Para salir pulsa `q` (*quit*).

Si el nombre del ejecutable es lo suficientemente descriptivo podemos matarlo directamente sin buscar su *pid* usando la orden `killall`:

```
pi@raspberrypi:~ $ killall -9 cat
pi@raspberrypi:~ $
```

Pero ten en cuenta que `killall` manda la señal a todos los procesos que tienen el campo `CMD` como el que se indica en el argumento.

Info Te proponemos el siguiente ejercicio. Imagina que has ejecutado el archivo `ejecutable` que tiene permiso `setuid` de `root`. Por un error de programación el programa no termina. ¿Qué orden deberás ejecutar para matarlo?

Control de versiones

El software, como cualquier proceso de ingeniería es un proceso iterativo. Los programas no se hacen de golpe, sino poco a poco, añadiendo una cosa cada vez. Después de añadir una función nueva es frecuente realizar un proceso de reingeniería, para dejar todo el programa más simple. En este proceso es frecuente generar *regresiones*, es decir, cosas que funcionaban dejan de funcionar. ¿Cómo volvemos a la situación anterior?

La solución, obviamente, es gestionando diferentes versiones. Pero gestionar de forma manual diferentes versiones es un proceso tedioso y muy propenso a error. Lo correcto es emplear un sistema de control de versiones. En este taller te proponemos GIT, el mismo que usa el núcleo Linux, y el mismo que usamos en la documentación y en el software de apoyo de este taller.

La orden `git` no es el programa más fácil de usar de tu nueva Raspberry Pi y no vamos a convertir este taller en un taller sobre *git*. Solo te comentaremos las órdenes que necesitas para actualizar el software del taller. Así siempre tendrás lo último.

Las carpetas `src` y `doc` de tu *home* son dos repositorios GIT que contienen lo mismo que en los repositorios oficiales de github.com. Fueron creados como se indica en el apéndice que describe [nuestra personalización de Raspbian](#). Cuando una carpeta es un repositorio GIT contiene una subcarpeta oculta llamada `.git`.

```
pi@raspberrypi:~ $ ls -d */.git
doc/.git src/.git
pi@raspberrypi:~ $
```

Este directorio tiene algunos archivos perfectamente legibles. En particular el archivo `config` :

```
pi@raspberrypi:~ $ cat src/.git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = https://github.com/FranciscoMoya/rpi-src.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = rigin
    merge = refs/heads/master
pi@raspberrypi:~ $
```

Aunque no entiendas ni la mitad lo que está claro es que ahí aparece la URL del repositorio original.

Para actualizar con los últimos cambios usa `git pull` :

```
pi@raspberrypi:~ $ cd src
pi@raspberrypi:~/src $ git pull
pi@raspberrypi:~ $ cd ../doc
pi@raspberrypi:~/doc $ git pull
pi@raspberrypi:~/doc $
```

Para reconstruir la documentación utiliza `gitbook` :

```
pi@raspberrypi:~/doc $ gitbook build
pi@raspberrypi:~/doc $
```

Cuando tengas algo de tiempo invierte en aprender acerca del control de versiones, especialmente de GIT. Cuando estés haciendo el *Trabajo Fin de Grado* será demasiado tarde. Empezarás a ponerte excusas a ti mismo y terminarás llevando el control de versiones de forma manual. Es una receta para el desastre, no digas que no te lo avisé. He visto casos de alumnos que estaban dispuestos a pagar más de 1000\$ para recuperar su precioso disco duro, donde estaba todo. He visto alumnos que corregían una y otra vez los mismos errores y siempre estaban ahí, como si se tratara de un fantasma. Pero sobre todo he vivido la diferencia entre el estudiante que hace software sabiendo que tiene el respaldo del control de versiones y el que piensa en no tocar lo que ya funciona. Si no se toca no se avanza. Si no se experimenta no aprende.

Actualmente hay un excelente libro [Citation not found] disponible gratuitamente [en línea](#) y en castellano. No hay excusas.

Los elementos de la RPi

La Raspberry Pi cuenta con un enorme número de periféricos y no nos va a dar tiempo a verlos todos. En las siguientes secciones veremos los conceptos fundamentales de la mayoría. En la página web del taller iremos añadiendo información que incorporaremos a ediciones futuras de este manual.

Entradas y salidas digitales

La versión online de este libro incorpora una línea de tiempo interactiva con la historia de la Raspberry Pi. Repasa un poco la evolución del concepto de la Raspberry Pi. Inicialmente parecía que iba a ser un pincho USB, similar a los mediacenters que se enchufan directamente en el televisor. Si el objetivo era reducir coste ¿por qué terminó siendo mucho más grande?

La respuesta la tenemos que buscar en los objetivos del proyecto. Sus diseñadores querían que fuera una plataforma docente, no simplemente un ordenador barato. Querían que los alumnos de la secundaria experimentaran por sí mismos no solo la programación, sino la construcción de sus propios equipos electrónicos. Era esencial que fuera fácil conectar cosas.

Por eso desde el primer modelo la Raspberry Pi cuenta con un buen número de pines que pueden configurarse como entradas o salidas digitales para controlar cualquier periférico, sensor o actuador externo. Se trata de los pines de *GPIO* (*General Purpose Input/Output*). En este capítulo trataremos sus características generales y posteriormente veremos los detalles de programación.

Comparación de los pines de E/S para los modelos originales y los modelos A+ y B+.

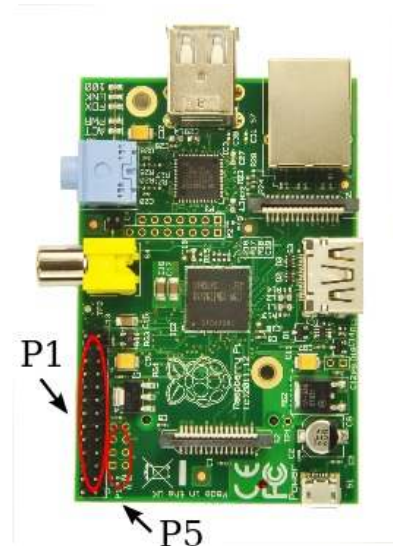
Conector de *GPIO*

Los modelos originales cuentan con un conector de 2x13 pines etiquetado como P1. Todos los modelos posteriores son compatibles con estos primeros 26 pines. Éste es el único pensado realmente para proporcionar entradas y salidas digitales de propósito general. El conector P1 proporciona acceso a 17 pines de *GPIO*. En la revisión 2 de la Raspberry Pi modelo B, junto a P1 se encuentra un zócalo desdoblado, el P5 de 2x4 pines, que proporciona acceso a 4 pines de *GPIO* adicionales.

Los modelos A+ y B+ amplían considerablemente el número de pines disponibles del conector P1 de 26 pines a un nuevo conector J8 de 40 pines. Entre ellos 9 pines más de *GPIO*. Sin embargo la compatibilidad es total, puesto que los 26 primeros pines mantienen su función original.

En lo sucesivo utilizaremos la numeración correspondiente al nuevo conector de 40 pines. Su equivalente en los conectores P1 y P5 puede verse en la tabla al inicio de este capítulo. Cuando hablemos de números de pines nos referiremos al nuevo conector J8 a menos que se indique lo contrario.

- Por un lado la mayoría de los pines son entradas/salidas digitales de propósito general. Pueden configurarse como entradas o salidas, pueden leerse o pueden escribirse con un valor digital, alto o bajo, uno o cero. Ten presente que el nivel alto es de 3.3V y no son tolerantes a tensiones de 5V.
- Los pines 8 y 10 pueden configurarse como interfaz UART para un puerto serie convencional. De hecho ésta es su configuración por defecto en Raspbian, ya que la UART se usa como consola.
- Por otro lado los pines 3 y 5, se pueden configurar como interfaz I2C para interactuar con periféricos que siguen este protocolo. En el taller ya lo hemos configurado de este modo.
- El pin 12 puede configurarse como salida PWM. En teoría los pines 12 y 13 pueden configurarse también como interfaz I2S (audio digital) pero hacen falta pines que no están disponibles fácilmente.
- Los pines 19, 21, 23, 24 y 26 se pueden configurar como la primera interfaz SPI (SPI0) para interactuar con periféricos que siguen este protocolo. En el taller ya los hemos configurado de este modo.
- Los pines 27 y 28 no están disponibles. Están reservados para la incorporación opcional de una memoria serie en las placas de expansión conforme a la especificación HAT. Son los únicos pines que en el arranque se configuran como salidas, todos los demás son configurados inicialmente como entradas para evitar problemas.
- Los pines 29, 31, 32, 33, 35, 36, 37, 38 y 40 proporcionan acceso a nuevas patas de GPIO que no estaban disponibles en los modelos originales. Estas patas pueden tener otros usos adicionales. Por ejemplo los pines 32, 33 y 35 pueden utilizarse para salidas PWM (solo dos canales disponibles). Además estas patas completan los pines necesarios para configurar otra interfaz SPI (SPI1), que no vamos a utilizar en el taller.



Configuración de los pines de E/S en los zócalos P1 y P5 de los modelos A y B revisión 2.

Warning

El zócalo P5 de los modelos originales está originalmente pensado para poblarlo desde la capa inferior del circuito. Los pines de la figura adjunta están consignados según este criterio. Si se monta en la capa superior la asignación de pines será su reflejo especular.

En total disponemos de 26 pines para entradas y salidas digitales y dos de ellos pueden usarse para control PWM.

Protección de GPIO

Cuando se utilizan los pines de *GPIO* para interfaz con hardware de cualquier tipo hay que poner mucho cuidado para no dañar la propia Raspberry Pi. Es muy importante comprobar los niveles de tensión y la corriente solicitada. Los pines de GPIO pueden generar y consumir tensiones compatibles con los circuitos de 3.3V (no son tolerantes a 5V) y pueden sacar hasta 16 mA. Eso es suficiente para iluminar un LED, pero para poco más.

Sin embargo hay que tener presente que la corriente que sale de esos pines proviene de la fuente de alimentación de 3.3V y esta fuente está diseñada para una carga pico de unos **3 mA por cada pin de GPIO**. Es decir, aunque el SoC de Broadcom permita drenar hasta 16 mA por cada pin la fuente no podrá dar mas de unos 78 mA en total (51 mA en los modelos originales). No supone riesgo alguno si intentas superar este límite, pero no funcionará.

Warning

Los pines GPIO de la Raspberry Pi no son tolerantes a tensiones de 5V. Están pensados para su utilización con circuitos de 3.3V y no tienen ningún tipo de protección. No debes drenar más de 16mA por pin.

Para evitar problemas se han fabricado una amplia variedad de tarjetas de expansión, que protegen de diversas formas los pines de GPIO. Las más conocidas son:

- [Pi-Face Digital](#)
- [Gertboard](#) de Gert van Loo, uno de los primeros voluntarios de la Raspberry Pi Foundation.

Una amplia variedad de métodos de protección está disponible en el tutorial de elinux.org titulado [GPIO Protection Circuits](#).

Programar entradas y salidas digitales

La referencia definitiva para programar cualquiera de los periféricos del BCM2835 es [la hoja de datos del fabricante](#) [Citation not found], aunque se trata de un documento denso y árido. Es también ilustrativo el documento de Gert van Loo [GPIO pads control](#).

En capítulos posteriores veremos algunos ejemplos para familiarizarnos con este periférico, pero es posible que te interese ampliar la información. Para empezar probablemente el mejor tutorial es el de elinux.org, que lleva por título [RPi Tutorial: Easy GPIO Hardware & Software](#) y especialmente los [ejemplos de programación utilizando diversos lenguajes y mecanismos](#). El [manual de la Gertboard](#) también tiene abundante información pero el software es difícil de encontrar.

Desde el punto de vista del programador los pines de GPIO de la Raspberry Pi se ven como dispositivos mapeados en memoria. Es decir, para configurar los pines, sacar valores digitales o leer señales digitales tenemos que leer o escribir en posiciones de memoria determinadas. Sin embargo el procesador de la Raspberry Pi utiliza un mecanismo denominado *memoria virtual*, en el que cada proceso ve un espacio de direcciones diferente, que no necesariamente tiene que corresponder con el espacio físico y que garantiza el aislamiento entre procesos. En GNU/Linux para poder acceder a direcciones físicas determinadas es necesario emplear un dispositivo (`/dev/mem`) que a todos los efectos se comporta como un archivo normal. Por ejemplo, en la posición `0x20200034` del dispositivo `/dev/mem` puede leerse el valor de las entradas GPIO0 a GPIO31.

Obviamente acceder a todo el espacio físico de direcciones es muy peligroso, puesto que permite que desde un proceso se pueda acceder a todo el espacio de direcciones de los demás procesos. No solo se compromete el aislamiento entre procesos sino también la seguridad del sistema. Un proceso malicioso podría utilizar funciones privilegiadas. Si piensas que eso no te afecta es que no sabes lo suficiente de seguridad informática. De vez en cuando echa un vistazo a las ponencias de [Blackhat](#) para ver qué se cuece en el mundo de la seguridad y verás que afecta a todo (robots, equipamiento médico, televisores, equipamiento industrial, domótica, sistemas de telecomunicación, ...). Un usuario malicioso podría incluso dañar físicamente la Raspberry Pi. Por este motivo el dispositivo `/dev/mem` tiene permisos de escritura solamente para el superusuario.

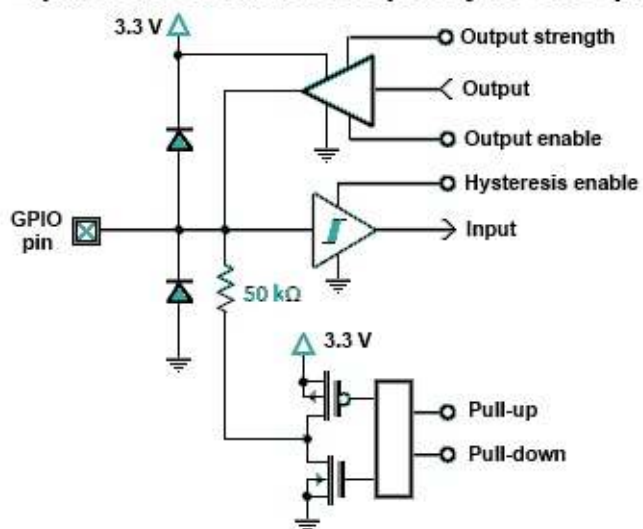
Las versiones recientes de Raspbian tienen un dispositivo `/dev/gpiomem` que permite acceder solo al rango de direcciones de los pines de GPIO y tiene permiso de escritura para el grupo `gpio`. El usuario `pi` es del grupo `gpio`. Por tanto los programas del usuario `pi` pueden actuar sobre los pines de GPIO. En la práctica esto puede no ser así porque muchas bibliotecas no utilizan aún `/dev/gpiomem`.

Características de los pines de GPIO

Los pines de GPIO de la Raspberry Pi incorporan un conjunto de características muy interesantes:

- Tienen capacidad de limitar el *slew rate*. Esto permitiría mejorar la inmunidad al ruido y reducir el ruido de *crosstalk*, pero a costa de alargar los tiempos de propagación.
- Se puede programar su *drive strength*, es decir, su capacidad de entregar corriente, entre 2 mA y 16 mA en saltos de 2 mA (ocho posibles valores). Básicamente consiste en la posibilidad de activar más o menos drivers en paralelo. Para mayor detalle consulta el documento de Gert van Loo referido arriba. Normalmente en el arranque está configurado a 8 mA. Esto no significa que no podamos pedir más corriente. Hasta 16 mA es seguro. Sin embargo si superamos los 8 mA la tensión bajará hasta el punto de que un uno lógico pueda dejar de interpretarse como uno y la disipación de calor será mayor. Por otro lado si programamos los pines a su máxima

Equivalent Circuit for Raspberry Pi GPIO pins



Circuito equivalente de un pin de GPIO (Fuente: [Mosaic Industries](#)).

capacidad tendremos picos de corriente que afectan al consumo y pueden llegar a afectar al funcionamiento de la tarjeta microSD, especialmente con cargas capacitivas. Este efecto se nota más cuanto mayor número de salidas conmuten simultáneamente.

- Es posible configurar las entradas con o sin *Schmitt trigger* de manera que la transición a nivel bajo y a nivel alto tengan umbrales diferentes. Esto permite dotar de cierta tolerancia a ruido.
- Es posible habilitar una resistencia de *pullup* y/o *pulldown*. Su valor es en torno a los 50KOhm.

La configuración de limitación de *slew rate*, *drive strength* y entrada con *Schmitt trigger* no se realiza pin a pin sino en bloques (GPIO0-27, GPIO28-45, GPIO46-53). Para los intereses del taller no debería haber problemas con la configuración por defecto.

Es posible examinar el estado y configuración de Para ello vamos a utilizar la utilidad `gpio` que incluye la biblioteca *wiringPi*.

```
pi@raspberrypi:~ $ gpio readall
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| BCM | wPi |   Name   | Mode | V | Physical | V | Mode |   Name   | wPi | BCM |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|  |  | 3.3v |  |  | 1 | 2 |  |  | 5v |  |  |
| 2 | 8 | SDA.1 | ALT0 | 1 | 3 | 4 |  |  | 5V |  |  |
| 3 | 9 | SCL.1 | ALT0 | 1 | 5 | 6 |  |  | 0v |  |  |
| 4 | 7 | GPIO.7 | IN | 1 | 7 | 8 | 1 | ALT0 | TxD | 15 | 14 |
|  |  | 0v |  |  | 9 | 10 | 1 | ALT0 | RxD | 16 | 15 |
| 17 | 0 | GPIO.0 | IN | 0 | 11 | 12 | 0 | IN | GPIO.1 | 1 | 18 |
| 27 | 2 | GPIO.2 | IN | 0 | 13 | 14 |  |  | 0v |  |  |
| 22 | 3 | GPIO.3 | IN | 0 | 15 | 16 | 0 | IN | GPIO.4 | 4 | 23 |
|  |  | 3.3v |  |  | 17 | 18 | 0 | IN | GPIO.5 | 5 | 24 |
| 10 | 12 | MOSI | ALT0 | 0 | 19 | 20 |  |  | 0v |  |  |
| 9 | 13 | MISO | ALT0 | 0 | 21 | 22 | 0 | IN | GPIO.6 | 6 | 25 |
| 11 | 14 | SCLK | ALT0 | 0 | 23 | 24 | 1 | OUT | CE0 | 10 | 8 |
|  |  | 0v |  |  | 25 | 26 | 1 | OUT | CE1 | 11 | 7 |
| 0 | 30 | SDA.0 | IN | 1 | 27 | 28 | 1 | IN | SCL.0 | 31 | 1 |
| 5 | 21 | GPIO.21 | IN | 1 | 29 | 30 |  |  | 0v |  |  |
| 6 | 22 | GPIO.22 | IN | 1 | 31 | 32 | 0 | IN | GPIO.26 | 26 | 12 |
| 13 | 23 | GPIO.23 | IN | 0 | 33 | 34 |  |  | 0v |  |  |
| 19 | 24 | GPIO.24 | IN | 0 | 35 | 36 | 0 | IN | GPIO.27 | 27 | 16 |
| 26 | 25 | GPIO.25 | IN | 0 | 37 | 38 | 0 | IN | GPIO.28 | 28 | 20 |
|  |  | 0v |  |  | 39 | 40 | 0 | IN | GPIO.29 | 29 | 21 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| BCM | wPi |   Name   | Mode | V | Physical | V | Mode |   Name   | wPi | BCM |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
pi@raspberrypi:~ $
```

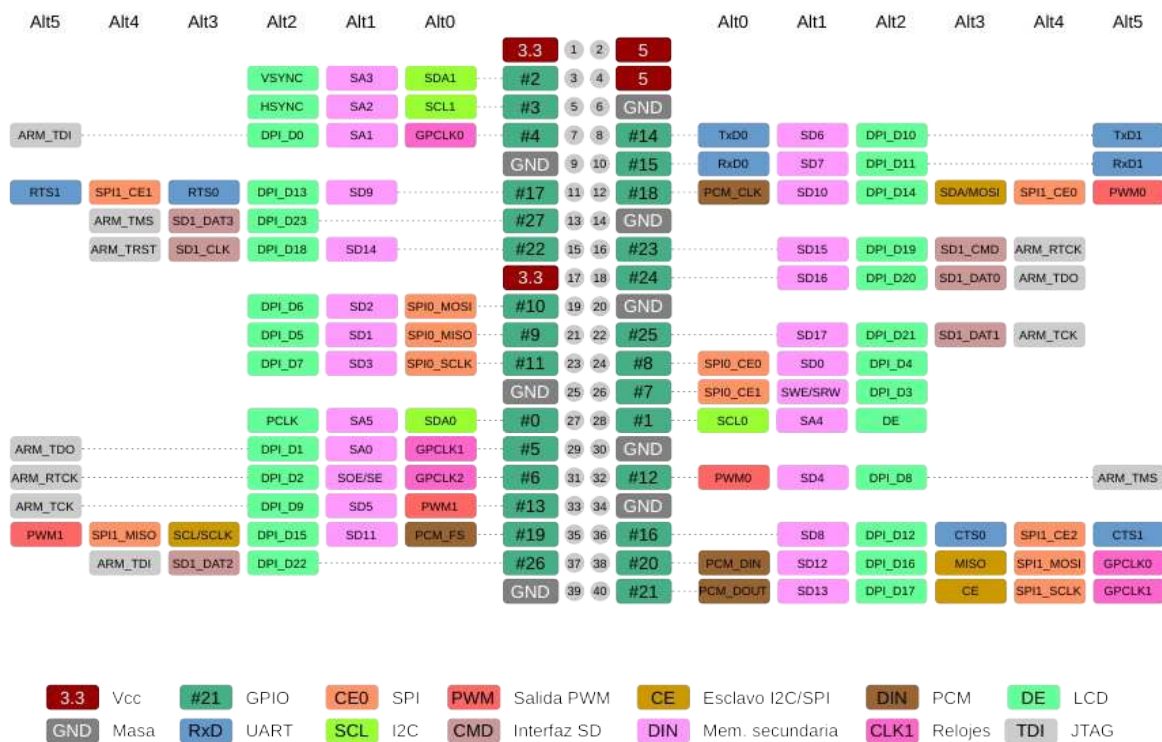
La columna *BCM* indica la numeración de Broadcom (*GPIO n*). La columna *Physical* contiene el número de pin en el conector J8 y la columna *V* contiene el valor leído. La columna *Name* representa la función del pin pero debes tener cuidado porque cuando aparece `GPIO.21` no se refiere a la nomenclatura de Broadcom, que es la usual, sino a una nomenclatura propia de la biblioteca *wiringPi*. Para interpretarlo correctamente debes mirar en el número correspondiente de la columna *BCM*, así que `GPIO.21` es en realidad *GPIO 5*. Esta confusión artificial ha sido objeto de numerosas críticas a la biblioteca *wiringPi* pero parece que los usuarios de Arduino lo ven como algo natural.

Funciones alternativas

Todos los pines de GPIO tienen la posibilidad de ser usados con otras funciones alternativas. Cada pin de GPIO puede configurarse como entrada, salida o como una de las seis funciones alternativas (desde *Alt0* hasta *Alt5*). No todas las configuraciones tienen sentido, consulta [Citation not found] para mayor detalle.

Junto a tu kit de iniciación en el taller recibirás una tarjeta que resume las funciones que a nosotros nos interesan. Para mayor detalle consulta el *flyer* que también te proporcionamos o la documentación de elinux.org que incluye toda la información dispersa. Si no encuentras ejemplos de alguno de los periféricos y no consigues hacerlo funcionar pregunta en el [foro del taller](#).

Este esquema puede resultarte útil para usos avanzados:



De todas formas debes tener presente que algunos elementos no pueden seleccionarse con seguridad porque ya se han utilizado en otras partes de la Raspberry Pi. Por ejemplo, el BCM2837 tiene dos interfaces SD y dos UART. Sin embargo ambas interfaces SD están ocupadas (una para la tarjeta microSD y otra para la comunicación WiFi). Análogamente UART0 se destina a la interfaz Bluetooth en la Raspberry Pi 3 y solo se expone UART1 en los pines 8 y 10. Otro ejemplo son los relojes de propósito general (*GPCLKx*) que permiten generar relojes de frecuencia programable en determinadas patas. *GPCLK1* está reservado para uso interno (Ethernet) y si se intenta usar lo más probable es que se cuelgue la *Raspberry Pi*. Nada grave, pero tampoco es agradable.

Manipulando pines en la consola

Vamos a empezar a usar los componentes sin escribir ni una línea de código, empleando programas que tienes disponibles. Conecta un LED a una de las patitas (por ejemplo GPIO18) con una resistencia para limitar la corriente a 15mA. Para ello te puedes ayudar de esta tabla tomada de theledlight.com corrigiendo el valor para los LEDs de Banggood que tenemos en el kit.

Tipo	Caída	Resistencia (15mA)
Rojo	1.7V	100 Ohm
Amarillo	2V	87 Ohm
Verde	2.1V	80 Ohm
Blanco	2.7V	40 Ohm
Azul	2.9V	27 Ohm

El kit tiene un conjunto de componentes discretos que incluye LEDs y resistencias. En teoría se incluyen tres de cada uno de los colores rojo, amarillo, verde y blanco. En la práctica dependiendo de la disponibilidad pueden incluir otro tipo de LEDs e incluso otro número. Por ejemplo, en un pedido de prueba que realizamos al comenzar el año recibimos 15 LEDs (5 rojos, 5 amarillos y 5 azules). Los objetivos del taller no se ven afectados en absoluto, así que esto nos parece una anécdota menor.

Los LEDs del kit no están coloreados, así que es difícil saber de qué color son. En las bolsitas suelen tener una letra (R, G, Y, W) para indicar el color, pero no siempre. De todas formas si ponemos una resistencia de 100 Ohm estamos seguros de no superar los límites y se enciende sin problemas hasta el LED azul. Veamos cómo encender y apagar el LED conectado

a la pata GPIO18:

```
pi@raspberrypi:~ $ gpio -g mode 18 out
pi@raspberrypi:~ $ gpio -g write 18 1
pi@raspberrypi:~ $ gpio -g write 18 0
pi@raspberrypi:~ $
```

En la primera línea hemos configurado la pata como salida. En las siguientes simplemente escribimos un valor en esa pata (1 y 0). La opción `-g` le indica a `gpio` que use la numeración de patas normal.

Nota Estamos de suerte en el kit de BangGood porque los diodos blancos tienen una caída de tensión moderada. Por desgracia, en muchas otras ocasiones esto no es así y es frecuente tener caídas de tensión de 3.4V en un LED blanco. Una consecuencia de esto es que no podemos encenderlos con salidas de 3.3V. En esos casos puedes usar el *level shifter* o un transistor, pero ¿es necesario? Aquí tienes el reto, si tuvieras un LED blanco con caída de 3.4V ¿qué harías? Usa los mismos componentes que antes pero configura el circuito para que se encienda sin problemas con una pata de GPIO.

El siguiente paso es utilizar las patas de GPIO como entradas digitales. Para ello conecta uno de los pulsadores entre otra pata de GPIO y masa. Lo normal es además poner una resistencia de *pull-up* de 10K entre la pata y 3.3V para que la entrada no este flotando mientras el pulsador no está apretado. Puedes hacerlo, pero te recordamos que también puedes usar el *pull-up* interno.

```
pi@raspberrypi:~ $ gpio -g mode 23 in
pi@raspberrypi:~ $ gpio -g mode 23 up
pi@raspberrypi:~ $ gpio -g read 23
1
pi@raspberrypi:~ $
```

Cada vez que ejecutemos `gpio -g read 23` nos devolverá el estado (0 para el conmutador *pulsado* y 1 para *no pulsado*).

Modulación de anchura de pulsos

Pulse Width Modulation (PWM) es una técnica que consiste en la variación del *duty cycle* de una señal digital periódica, fundamentalmente con dos posibles objetivos:

- Por un lado se puede utilizar como mecanismo para transmitir información. Por ejemplo, los servo-motores tienen una entrada digital por la que se transmite el ángulo deseado codificado en PWM.
- Por otro lado se puede utilizar para regular la cantidad de potencia suministrada a la carga. Por ejemplo, las luminarias LED frecuentemente utilizan reguladores PWM para permitir el control de intensidad.

La Raspberry Pi tiene una varias patas de GPIO (GPIO12, GPIO13, GPIO18 y GPIO19) que puede configurarse como salida de alguno de los dos canales PWM. El propio BCM2835 se encarga de gestionar la generación de la señal, liberando completamente al procesador principal.

El periférico PWM de la Raspberry Pi es muy flexible pero solo dispone de dos canales (*PWM0* y *PWM1*). Puede funcionar en modo PWM o en modo serializador. En el modo serializador simplemente saca por la pata correspondiente los bits de las palabras que se escriben en un *buffer*. Veamos primero el modo PWM.

El usuario puede configurar dos valores:

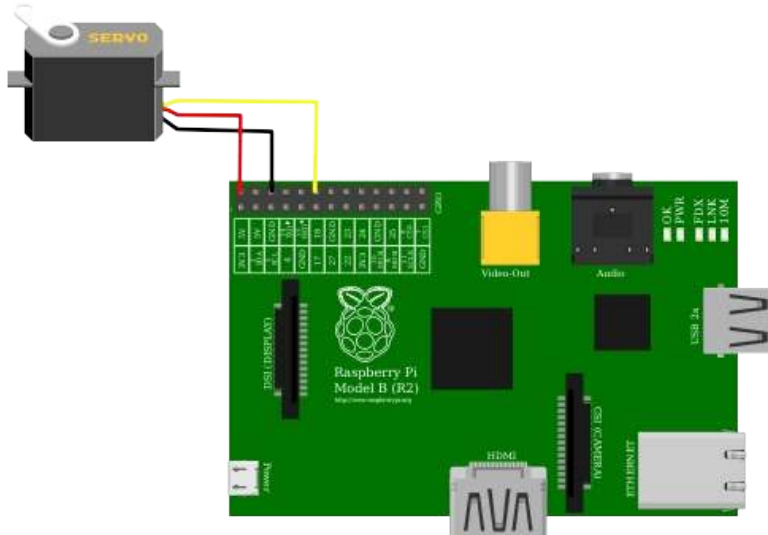
- Un *rango* de valores disponible (hasta 1024).
- Un *valor* que determina el *duty cycle*. El módulo PWM se encarga de mantener el *duty cycle* en la relación valor/rango.

La frecuencia base para PWM en Raspberry Pi es de 19.2Mhz. Esta frecuencia puede ser dividida mediante el uso de un divisor indicado con `pwmSetClock`, hasta un máximo de 4095. A esta frecuencia funciona el algoritmo interno que genera la secuencia de pulsos, pero en el caso del BCM2835 se dispone de dos modos de funcionamiento, un modo equilibrado (*balanced*) en el que es difícil controlar la anchura de los pulsos, pero permite un control PWM de muy alta frecuencia, y un modo *mark and space* que es mucho más intuitivo y más apropiado para controlar servos. El modo *balanced* es apropiado para controlar la potencia suministrada a la carga.

En el modo *mark and space* el módulo PWM incrementará un contador interno hasta llegar a un límite configurable, el rango de PWM, que puede ser de como máximo 1024. Al comienzo del ciclo el pin se pondrá a 1 lógico, y se mantendrá hasta que el contador interno llegue al *valor* puesto por el usuario. En ese momento el pin se pondrá a 0 lógico y se mantendrá hasta el final del ciclo.

Veamos su aplicación al control de un servomotor. Un servomotor tiene una entrada de señal para indicar la inclinación deseada. Cada 20ms espera un pulso y la anchura de este pulso determina la inclinación del servo. Alrededor de 1.5ms es la anchura del pulso necesaria para la posición centrada. Una anchura menor hace girar el servo en sentido antihorario (hasta 1ms aproximadamente) y una duración mayor lo hace girar en sentido horario (hasta 2ms aproximadamente). En este caso hay que calcular el rango y el divisor para que el pulso se produzca cada 20ms y el control de la anchura del pulso alrededor de los 1.5ms sea con la máxima resolución posible.

El montaje es tal como muestra la figura. El cable rojo del servo (V+) se conecta a +5V en P1-2 o P1-4, el cable negro o marrón (V-) a GND en P1-6 y el cable amarillo, naranja o blanco (signal) a GPIO18 en P1-12. No se necesita ningún otro componente.



Montaje de un microservo para ser controlado directamente desde GPIO18 configurado como salida PWM.

Para tener máximo control de la posición del servo probaremos con el rango máximo, de 1024. En ese caso el divisor tiene que ser tal que la frecuencia del pulso PWM sea:

$$f = \frac{f_{base}}{\text{rango} \times \text{div}} = \frac{19.2 \times 10^6 \text{ Hz}}{1024 \times \text{div}} = \frac{1}{20 \text{ ms}} = 50 \text{ Hz}$$

Es decir, el divisor habría que configurarlo a 390. El rango completo del servo depende del modelo concreto. Teóricamente debería ser entre 52 y 102, siendo el valor completamente centrado 77. En la práctica habrá que probar el servo concreto porque los límites de 1ms y 2ms no son estrictos. Nuestros experimentos dan un rango útil entre 29 y 123 para el microservo de TowerPro disponible en el kit del alumno.

Veamos cómo se puede controlar sin programar nada. Primero configuramos la pata GPIO18 como habíamos calculado:

```
pi@raspberrypi:~ $ gpio -g mode 18 pwm
pi@raspberrypi:~ $ gpio pwm-ms
pi@raspberrypi:~ $ gpio pwmr 1024
pi@raspberrypi:~ $ gpio pwmc 390
pi@raspberrypi:~ $
```

Ahora podemos cambiar el valor correspondiente para que gire a la posición deseada.

```
pi@raspberrypi:~ $ gpio -g pwm 18 52
pi@raspberrypi:~ $ gpio -g pwm 18 102
pi@raspberrypi:~ $
```

Warning Recuerda que solo hay dos canales PWM disponibles y que estos canales solo se pueden asignar a ciertas patas (PWM0 en GPIO12 y GPIO18, PWM1 en GPIO13 y GPIO19).

Más sobre PWM

Para los propósitos del taller no nos detendremos más en el módulo PWM, pero tened en cuenta que la gama de posibilidades es mucho mayor. No descartamos incorporar a esta sección en el futuro más información sobre los otros modos de funcionamiento. De momento preferimos avanzar para conocer otros periféricos.

Ten presente que el audio analógico incluido en la Raspberry Pi utiliza los dos canales PWM, así que si usas sonido analógico procura evitar el uso de PWM. Nuestra recomendación es que uses un altavoz Bluetooth o bien puedes emplear una placa de audio externa (*HiFi-Berry*, por ejemplo).

Comunicaciones I2C

I2C [Citation not found] es un bus de comunicaciones entre circuitos integrados desarrollado por Phillips Semiconductors (ahora NXP Semiconductors). Se trata de un bus muy sencillo con solo dos hilos, una línea de datos (SDA) y una línea de reloj (SCL). Se pueden realizar transmisiones serie bidireccionales de hasta 100 kbit/s en modo estándar y 400 kbit/s en modo rápido. Las versiones más modernas de I2C incorporan modos con mayores tasas de transferencia (hasta 5Mbit/s) pero el procesador de la Raspberry Pi no los implementa.

La Raspberry Pi dispone de dos periféricos para implementar I2C [Citation not found], el BSC (*Broadcom Serial Controller*) que implementa el modo maestro, y el BSI (*Broadcom Serial Interface*) que implementa el modo esclavo. Describiremos únicamente el BSC, que tiene mucho mayor interés para este taller.

El BSC implementa tres maestros independientes que tienen que estar en buses I2C separados (no permite multi-maestro). Sin embargo BSC0 se reserva para la identificación de las placas de expansión (especificación HAT, pines 27 y 28) y BSC2 es de uso exclusivo para la interfaz HDMI. Usaremos por tanto habitualmente BSC1, mediante las patas 3 y 5 del conector J8.

La interfaz de programación es, como en todos los periféricos de la Raspberry Pi, un conjunto de registros mapeados en memoria. Sin embargo en este caso se dispone de un *driver* del kernel del sistema operativo que nos simplifica significativamente la vida.

Explorar el bus

Los dispositivos conectados a un bus I2C tienen una dirección de 7 bits. Aunque existe la posibilidad de utilizar direcciones de 10 bits lo cierto es que es bastante raro encontrar dispositivos con direcciones de más de 7 bits. En este taller asumiremos direcciones de 7 bits. Esto implica que como máximo podemos tener 117 dispositivos conectados (algunas direcciones están reservadas).

Lo primero que podemos hacer es descubrir qué interfaces I2C tenemos disponibles. En los modelos originales solo se disponía de BSC0, mientras que en los nuevos BSC1 se usa para la identificación de placas de expansión HAT (*hardware attached on top*). Puedes examinarlo tú mismo ejecutando `i2cdetect` :

```
pi@raspberrypi:~ $ i2cdetect -l
i2c-1    i2c          3f804000.i2c          I2C adapter
pi@raspberrypi:~ $
```

Solo nos aparece una interfaz I2C (`i2c-1`). Ahora podemos mirar qué dispositivos hay conectados en el bus. Conecta el módulo MPU6050 que viene incluido en el kit. Solo hay que conectar *GND*, *VCC* a 3.3V, *SDA* y *SCL*. Volveremos a usar `i2cdetect` indicando ahora el número de bus I2C disponible:

```
pi@raspberrypi:~ $ i2cdetect -y 1
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  68  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
pi@raspberrypi:~ $
```

Solo se ve un dispositivo conectado con dirección 68 en hexadecimal (`0x68`). En realidad el MPU6050 puede tener dos direcciones configurando la patita *AD0*. Normalmente esa patita tiene un pull-down que la pone a cero lógico, que corresponde a la dirección `0x68` , pero podemos conectarla a 3.3V para tener la dirección `0x69`. Esto permite tener dos MPU6050 en el mismo bus. Uno con dirección `0x68` y otro con dirección `0x69`.

Ya podemos leer y escribir en cualquiera de los dispositivos I2C conectados. Para leer podemos utilizar `i2cget` y para escribir podemos usar `i2cset` :

```
pi@raspberrypi:~ $ i2cget -y 1 0x68 117
0x68
pi@raspberrypi:~ $
```

Hemos leído el registro 117 del dispositivo con dirección 0x68 del bus i2c-1. El MPU6050 es un [acelerómetro y giróscopo de InvenSense](#) que mide también la temperatura y puede combinarse de forma directa con un magnetómetro para tener una IMU (*Inertial Measurement Unit*) completa.

Si miramos el [mapa de registros](#) el registro 117 corresponde a *Who Am I*, es un registro que simplemente devuelve la dirección base del dispositivo. Puede usarse para asegurarnos de que el dispositivo está en la dirección esperada y está respondiendo. Es un registro de solo lectura, así que si intentamos escribir en él lo ignorará completamente:

```
pi@raspberrypi:~ $ i2cset -y 1 0x68 117 0x70
pi@raspberrypi:~ $ i2cget -y 1 0x68 117
0x68
pi@raspberrypi:~ $
```

Vamos a leer la última medida de la temperatura. La medida está en los registros 65 y 66, pero si leemos los dos registros de forma independiente podemos estar leyendo parte de la temperatura de una medida y otra parte de otra medida. Por eso hay que leer los dos registros de golpe, haciendo una transferencia de 16 bits (modo palabra):

```
pi@raspberrypi:~ $ i2cget -y 1 0x68 65 w
0x0000
pi@raspberrypi:~ $
```

Ups, un valor extraño. Algo falla. Lo normal es tomarse un poco de tiempo en leer detenidamente la hoja de datos y el mapa de registros, para entender cómo funciona. La pista del problema nos llega en cuanto intentamos leer el registro 107 (*Power Management 1*):

```
pi@raspberrypi:~ $ i2cget -y 1 0x68 107
0x40
pi@raspberrypi:~ $
```

Eso significa que está puesto a uno el bit 6, que curiosamente corresponde con el modo *SLEEP*. El dispositivo arranca en modo *dormido* para no consumir batería, tenemos que despertarlo quitando ese bit:

```
pi@raspberrypi:~ $ i2cset -y 1 0x68 107 0
pi@raspberrypi:~ $ i2cget -y 1 0x68 65 w
0x90f2
pi@raspberrypi:~ $
```

Esto ya es otra cosa, ya tenemos lecturas de temperatura. Pero es muy importante interpretarlas bien. I2C es un protocolo orientado a bytes. Se transfiere byte a byte. Al transferir una palabra por I2C el ordenador entiende que primero llega el byte bajo y luego el alto. Este convenio se llama *little-endian* y es el dominante hoy en día. Sin embargo si miramos el mapa de registros veremos que la dirección 65 corresponde al byte alto y el 66 al byte bajo. ¡Por tanto los bytes están cambiados! La lectura hay que interpretarla como 0xf290. Vamos a ver cómo podemos hacerlo con la *shell*:

```
pi@raspberrypi:~ $ T=$(i2cget -y 1 0x68 65 w)
pi@raspberrypi:~ $ echo "0x${T:4:2}${T:2:2}"
0xf290
pi@raspberrypi:~ $
```

Lo que saca la orden `i2cget` lo metemos en una variable `T` y luego imprimimos un `0x` seguido de los dos caracteres a partir del cuarto de `T` y seguido de los dos caracteres a partir del segundo de `T`. Esto empieza a complicarse, ¿verdad? Va siendo hora de pensar en hacer un programa en C o en Python. Pero ya aunque sea por dejar el ejemplo completo vamos a hacer la transformación completa a una temperatura en grados Celsius:

```
pi@raspberrypi:~ $ echo "$(($T - 0x10000))/340 + 36.53" | bc -l
26.130000000000000000000000
pi@raspberrypi:~ $
```

Componemos la fórmula que viene en el mapa de registros. La temperatura es el valor del registro correctamente interpretado partido por 340 mas 36.5. El problema es que la *shell* no sabe hacer operaciones aritméticas con reales, así que se lo dejamos a otro programa, en este caso `bc`. A `bc` le pasamos la expresión a calcular, que es $0xf230/340 + 36.5$. El problema es que `bc` no entiende de números hexadecimales, así que lo tenemos que pasar a decimal con signo. Eso se puede hacer con el operador `$(())` de la *shell*, que sirve para calcular operaciones sencillas con enteros. Como es negativo su valor es el resultado de restar `0x10000`.

¿Demasiado complicado? Sí, yo también lo creo. Por eso es necesario que simplifiquemos esto con el mecanismo más poderoso que nos ofrece la informática: la *abstracción*. Pero dejaremos esto para el momento en que veamos cómo implementar todas estas lecturas en C o en Python.

Vamos a terminar este capítulo ilustrando otra forma de leer, de una ráfaga, todos los registros de medida:

```
pi@raspberrypi:~ $ i2cdump -y -r 59-72 1 0x68 b
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f      0123456789abcdef
30:                                fd f4 fe 5c 3e          ???\>
40: f0 f1 60 ff a2 ff 39 ff 78          ??`.?.9.x
pi@raspberrypi:~ $
```

La orden `i2cdump` permite leer de una ráfaga un conjunto de registros. Leemos de golpe desde el registro 61 hasta el 72. Corresponden a las lecturas de (por orden):

- Acelerómetro.
 - Aceleración en X: `0xfdf4`
 - Aceleración en Y: `0xfe5c`
 - Aceleración en Z: `0x3ef0`
- Temperatura: `0xf160`.
- Giróscopo:
 - Rotación en X: `0xffa2`
 - Rotación en Y: `0xff39`
 - Rotación en Z: `0xff78`

Solo queda interpretar estos números como enteros con signo en complemento a 2. Queda propuesto como ejercicio.

El giróscopo mide la velocidad de giro en los tres ejes. Obviamente si no se dispone de una referencia global se irá acumulando error y su medida absoluta no será muy útil para determinar la orientación real. Por eso el MPU6050 tiene la opción de acoplarse con un magnetómetro, que proporciona una referencia real del norte, que permite corregir los errores acumulados. En el taller no vamos a usar magnetómetro, pero plantéatelo para tus proyectos.

Comunicaciones SPI

SPI es un protocolo alternativo a I2C que utilizan muchos dispositivos. Hoy en día es frecuente encontrar dispositivos que implementan tanto I2C como SPI.

La elección de cuál utilizar depende de la aplicación:

- I2C utiliza menos pines y permite direccionar muchos más dispositivos.
- SPI utiliza más pines pero permite una velocidad muy superior.

Al igual que I2C es un protocolo serie, pero la dirección de los dispositivos no se transmite por el canal de datos sino que se utilizan patas específicas para seleccionarlos. Al igual que en I2C hay un maestro que toma el rol activo en la comunicación y uno o varios esclavos que asumen el rol pasivo.

Cada interfaz SPI cuenta con al menos cuatro patas:

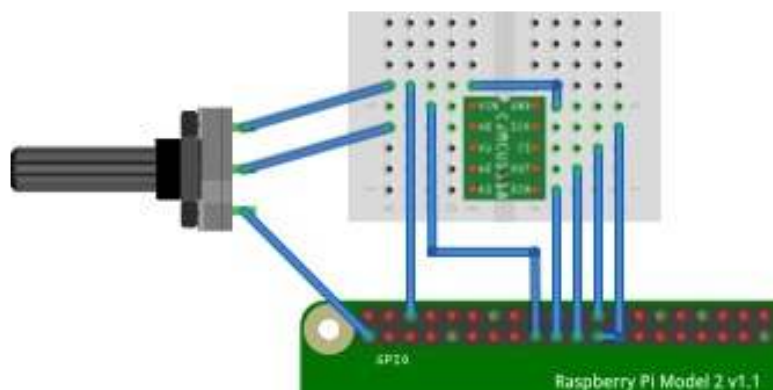
- **SCLK** (*Serial CLock*). Señal de reloj respecto a la que se sincronizan el resto de las señales.
- **MISO** (*Master In, Slave Out*). Entrada de datos para el maestro, salida de datos para el esclavo.
- **MOSI** (*Master Out, Slave In*). Salida de datos para el maestro, entrada de datos para el esclavo.
- **CE_n** (*Chip Enable*). Una o varias señales de selección de destino activas a nivel bajo. Para enviar o recibir del dispositivo cero se activa la señal CE₀, para el dispositivo uno se activa CE₁, etc.

En la Raspberry Pi disponemos de tres periféricos independientes para implementar el modo esclavo y el maestro. El modo esclavo lo incorpora en el mismo BSI (*Broadcom Serial Interface*) que implementa también el esclavo de I2C. En el taller veremos el modo maestro, que tiene mucho más interés práctico.

La Raspberry Pi cuenta con tres interfaces maestro SPI (SPI0, SPI1 y SPI2) aunque solo una de ellas (SPI0) es visible para los modelos originales y dos de ellas (SPI0 y SPI1) para los modelos con conector J8 de 40 pines. Cada una de estas interfaces cuenta con dos líneas de selección de destino (*CE0* y *CE1*). De ellas SPI0 es más evolucionada, puesto que permite DMA (acceso directo a memoria). SPI0 está diseñada para transferencias de alta velocidad (reloj de hasta 125Mhz) sin producir una carga significativa para el procesador. Sin embargo SPI1 no tiene posibilidad de usar DMA y solo dispone de una pequeña FIFO de cuatro palabras de 32 bits.

El kit del alumno dispone de un conversor analógico-digital con interfaz SPI CJMCU-1118, que incorpora el Texas Instruments ADS1118. Se trata de un conversor analógico-digital de 16 bits con amplificador de ganancia programable y con un sensor de temperatura. Puede medir cuatro señales analógicas referidas a masa o dos señales analógicas diferenciales. Vamos a ilustrar el uso de SPI con este módulo.

Para ello tendremos que estudiar bien su [hoja de datos](#) antes de trabajar con él. De momento para familiarizarnos con el dispositivo conectaremos un potenciómetro de 10K como muestra la figura.



Montaje de un potenciómetro de 10KOhm como fuente analógica para el CJMCU-1118.

Para comunicarnos con un dispositivo SPI tenemos que configurar primero una serie de parámetros:

- La polaridad de la señal *chip select* (**CSPOL**). Normalmente es activa a nivel bajo y no necesita modificarse.
- La polaridad del reloj (**CPOL**). Un valor 0 significa que el nivel de descanso del reloj es bajo. Un valor 1 significa que el nivel de descanso del reloj es alto.
- La fase del reloj (**CPHA**). Si tiene un valor 0 significa que las transiciones de *SCLK* ocurren en la mitad de cada bit de datos transmitido. Un valor 1 significa que las transiciones de *SCLK* ocurren al principio de cada bit.

- Frecuencia del reloj. Se configura seleccionando una fuente de reloj y un divisor (**CDIV**). La fuente de reloj normalmente es de 125MHz.

Las diferentes combinaciones de CPOL y CPHA dan lugar a los cuatro modos posibles:

Modo	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

De la hoja de datos del ADS1118 podemos ver que el dispositivo es compatible con el modo 1 (CPOL=0, CPHA=1), la polaridad de CE es también activa baja y admite un reloj de hasta 4MHz.

Para interactuar con él vamos a usar las herramientas incluidas en la biblioteca *pigpio*. En particular usaremos `pigpiod` y su interfaz de órdenes `pigs`.

Primero ejecutamos el servidor `pigpiod`:

```
pi@raspberrypi:~ $ sudo pigpiod
pi@raspberrypi:~ $ _
```

Utilizamos `sudo` para que se ejecute con permisos de superusuario. Se llama servidor porque atiende peticiones de los clientes. Por sí mismo no hace nada, sino que espera a que un cliente le solicite operaciones concretas.

El cliente se llama `pigs` y puede ejecutarse como usuario normal. Por ejemplo:

```
pi@raspberrypi:~ $ pigs help
...
pi@raspberrypi:~ $ _
```

La lista de órdenes que admite es muy larga, pero nos quedaremos con las relativas a SPI:

Orden	Significado
<code>spio c b f</code>	Abre el canal SPI <i>c</i> usando un reloj de frecuencia <i>b</i> y con las banderas especificadas en <i>f</i> . Devuelve un número (<i>handle</i>) que debe usarse en todas las demás órdenes.
<code>spir h num</code>	Lee <i>num</i> bytes del canal SPI asociado al handle <i>h</i> .
<code>spiw h bvs</code>	Escribe bytes en el canal SPI asociado al handle <i>h</i> .
<code>spix h bvs</code>	Escribe bytes en el canal SPI asociado al handle <i>h</i> y a la vez recibe el mismo número de bytes por el mismo canal.
<code>spic h</code>	Cierra el canal SPI asociado al handle <i>h</i> .

La orden `spio` tiene un parámetro `f` que corresponde a una serie de banderas (*flags*) de configuración, que siguen el siguiente formato de 22 bits:

21-16	15	14	13-10	9	8	7-5	4-2	1-0
bbbbbb	R	T	nnnn	W	A	uuu	ppp	mm

Donde cada uno de los campos significa lo siguiente:

Campo	Significado
<i>mm</i>	Modo SPI.
<i>ppp</i>	Polaridad de <i>CEi</i> . 0 = activa baja, 1 = activa alta.
<i>uuu</i>	Uso de <i>CEi</i> . 0 = se usa, 1 = no se usa.
<i>A</i>	SPI auxiliar. 0 = SPI normal, 1 = SPI auxiliar.
<i>W</i>	Número de cables. 0 = 4 cables (normal), 1 = 3 cables.
<i>nnnn</i>	Número de bytes a escribir antes de cambiar MOSI a MISO (solo para <i>W</i> = 1).
<i>T</i>	Orden de bits transmitidos. 0 = MSb primero, 1 = LSb primero.
<i>R</i>	Orden de bits recibidos. 0 = MSb primero, 1 = LSb primero.
<i>bbbbbb</i>	Tamaño de palabra en bits (0-32). 0 = 8 bits. Solo para SPI auxiliar.

Tenemos suerte porque la mayoría de los parámetros por defecto corresponden a los valores que necesita el módulo CJMCU-1118. Tan solo hay que cambiar el modo SPI.

```
pi@raspberrypi:~ $ pigs spio 0 4000000 1
0
pi@raspberrypi:~ $ _
```

Ahora tenemos que configurar el módulo con las entradas referidas a masa, en modo continuo a 128 SPS y con rango de medida (FSR) de $\pm 2.048V$. El ADS1118 escribe su configuración a la vez que lee datos. Si no se desea escribir el registro de configuración simplemente hay que escribir ceros. Este es el formato del registro de configuración:

15	14-12	11-9	8	7-5	4	3	2-0
SS	MUX	PGA	MODE	DR	TS_MODE	PU	NOP

Donde cada campo corresponde a lo siguiente:

Campo	Significado
SS	A 1 para empezar una conversión cuando está en modo <i>single shot</i> .
MUX	Configura el multiplexor de la entrada de conversor. Para señales referidas a masa es <i>1xx</i> con <i>xx</i> correspondiente a la entrada que se desea.
PGA	Define el rango a plena escala del amplificador de ganancia programable. <i>000</i> para $\pm 6.144V$, <i>001</i> para $\pm 4.096V$, <i>010</i> para $\pm 2.048V$, <i>011</i> para $\pm 1.024V$, <i>100</i> para $\pm 0.512V$ y el resto para $\pm 0.256V$.
MODE	0 para <i>modo continuo</i> , 1 para modo <i>single shot</i> .
DR	Tasa de muestreo. Desde 8 hasta 860 SPS. $8 \cdot 2^n$ SPS.
TS_MODE	0 = Modo ADC, 1 = modo sensor de temperatura.
PU	Habilita <i>pull-up</i> en DOUT.
NOP	Siempre <i>xx1</i> . <i>011</i> = Actualizar registro de configuración.

En nuestro caso escribimos `0100 0100 1000 1011`.

```
pi@raspberrypi:~ $ pigs spix 0 0x44 0x8b 0x44 0x8b
4 0 0 68 139
pi@raspberrypi:~ $ _
```

Para leer el valor del potenciómetro basta volver a repetir la orden pero en este caso no es necesario enviar un registro de control válido.

```
pi@raspberrypi:~ $ pigs spix 0 0 0 0 0
4 75 230 68 139
pi@raspberrypi:~ $
```

Enviando cuatro octetos devuelve cuatro octetos: la última muestra y un eco del registro de control.

Para interpretar la muestra hay que imprimir los dos primeros octetos como un número de 16 bits en complemento a 2 y multiplicar el resultado por los voltios que suponen cada bit (rango a plena escala dividido por el valor máximo, es decir, 2.048/32768).

```
pi@raspberrypi:~ $ V=$(printf "0x%02x%02x" 75 230)
pi@raspberrypi:~ $ echo "$(($V))/32768*2.048" | bc -l
1.214375000000000000000000
pi@raspberrypi:~ $
```

Con el programa `printf` podemos imprimir valores de forma similar a como lo hace la función `printf` de C. El operador `$(orden)` de la *shell* simplemente devuelve la salida estándar de la orden que encierra. En este caso la utilizamos para meter la muestra en hexadecimal en la variable `v`. A continuación usamos el operador `$((expr))` de la *shell* para transformar ese número hexadecimal en decimal y componemos una expresión que es calculada por `bc`.

Comunicación con UART

UART son las siglas de *Universal Asynchronous Receiver Transmitter*. Se trata de la interfaz de comunicación serie dominante en los microcontroladores de gama baja y en los ordenadores antiguos. Antes de USB la UART era el mecanismo para conectar el teclado, el ratón o incluso el modem de comunicaciones. Antes aún era la interfaz empleada por los terminales o consolas para interactuar con un ordenador. El programa Terminal que ejecutas en tu Raspberry Pi no es sino una emulación de estos terminales antiguos.

Hoy en día es casi imposible encontrar una de estas interfaces serie en un ordenador, y prácticamente ha sido reemplazada por USB. Afortunadamente tenemos cables USB especiales que incorporan un adaptador UART-USB. En el kit del alumno de este taller tienes uno. Este adaptador te permite comunicar la UART de la Raspberry Pi con un puerto USB cualquiera, ya sea de un ordenador o de otra Raspberry Pi. Esto permite utilizar la Raspberry Pi sin necesidad de disponer de un monitor y sin necesidad de tener conexión de red ni teclado USB. Por ejemplo, desde otra Raspberry Pi o desde un portátil.

El puerto serie ocupa los pines GPIO14 (Tx) y GPIO15 (Rx). Por defecto *Raspbian* utiliza estos pines para una consola serie. El SoC de Broadcom soporta dos UARTs pero ambas se configuran en los mismos pines de la cabecera J8. UART0 es un periférico estándar de ARM (PL011) e implementa todas las capacidades esperables en una UART de alto rendimiento. UART1 implementa una versión simplificada que se denomina mini-UART por el fabricante. Está pensada para ser usada con dispositivos de baja tasa de transferencia, como la consola.

En todos los modelos de Raspberry Pi salvo en el Compute Module solo disponemos de los pines GPIO14 y GPIO15 para ambas UART. Por este motivo *Raspbian* utiliza UART0 para la consola y no se suele emplear UART1.

Pero claro, eso significa que la UART está ocupada en la consola. Es necesario quitar la consola para poder emplear la UART en otros fines. Para desactivar la consola hay que editar el archivo `/boot/cmdline.txt` como superusuario, eliminar el fragmento que dice `console=serial0,115200`, y reiniciar la Raspberry Pi.

Para poder utilizar los pines GPIO14 y GPIO15 como pines de entrada/salida digital utiliza la herramienta de configuración y en la pestaña *Interfaces* desactiva la opción *Serial*.

Pero volvamos a su uso como consola, que es muy útil cuando quieras poner tu Raspberry Pi en un robot u otro tipo de equipos donde enchufar un monitor es impensable. Es posible conectar un cable USB a estos pines para conectarse a la Raspberry Pi sin necesidad de ningún tipo de configuración de red. En el kit del alumno habrás recibido un cable USB en un extremo y con cuatro conectores Dupont hembra en el otro. Puedes conectarlo de la siguiente manera:

- El cable rojo es la alimentación del USB (5V). La Raspberry Pi ya tiene su propia alimentación, así que este cable **no debes conectarlo**.
- El cable negro es la masa, conéctalo al pin J8-6.
- El cable blanco es el de transmisión de datos desde la UART al puerto USB. Conéctalo al pin J8-8 (GPIO14).
- El cable verde es el de recepción de datos desde el puerto USB hacia la UART. Conéctalo al pin J8-10 (GPIO15).

Con esto debería ser suficiente para tener la consola funcionando en cualquier modelo de Raspberry Pi anterior a la 3B. El problema es que en la Raspberry Pi 3 la UART se utiliza para la interfaz Bluetooth y la consola se configura con la mini-UART. Esto tiene importantes consecuencias que no están del todo resueltas aún. Como usuario no vas a enfrentarte en absoluto a los problemas que plantea, pero ten presente que para que la mini-UART funcione correctamente hay que fijar la frecuencia del core a 250 MHz, no es posible modular la frecuencia para ahorrar consumo.

Probar la consola es muy sencillo, utilizando la propia Raspberry Pi. Conecta también el extremo USB del cable a un puerto USB libre y ejecuta:

```
pi@raspberrypi:~ $ miniterm.py /dev/ttyUSB0 115200
Raspbian GNU/Linux 8 raspberrypi ttyS0

raspberrypi login:
```

Entra como usuario `pi` y contraseña `raspberrypi`. Como ves dispones de una consola completamente funcional en la UART. Puedes conectarte a ella con cualquier portátil y usar la Raspberry Pi sin necesidad de un monitor y un teclado. Para equipos móviles es una gran ventaja. Para salir de `miniterm.py` presiona la tecla *Ctrl*, *AltGr* y *J*.

Queda fuera de los objetivos del taller explicar cómo usar la consola desde un portátil con Windows. Si dispones de un GNU/Linux en tu portátil el sistema es el mismo que hemos explicado. Instala la herramienta `screen` y úsala igual que hemos usado `miniterm.py`. En ese caso se sale con *Ctrl-A* seguido de `\`.

Comunicaciones en red

Los modelos de Raspberry Pi más utilizados (B, B+, 2B y 3B) incorporan interfaz Ethernet. El nuevo modelo 3B incorpora además una interfaz WiFi y la mayoría de los usuarios de otros modelos utilizan un pincho USB WiFi (*dongle WiFi*) para actualizar el sistema. Por tanto es obligado que hablemos un poco de comunicaciones, aunque lo haremos desde una perspectiva puramente práctica, sin entrar en detalles excesivamente técnicos. Por supuesto te recomendamos que amplíes la información con la amplia bibliografía existente, por ejemplo [Citation not found].

Abre un terminal de órdenes y teclea lo siguiente:

```
pi@raspberrypi:~ $ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN group default qlen 1000
    link/ether b8:27:eb:0c:0c:03 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::84bc:c6c6:4e28:fc2a/64 scope link tentative
        valid_lft forever preferred_lft forever
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:13:ef:71:03:d0 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.38/24 brd 192.168.1.255 scope global wlan0
        valid_lft forever preferred_lft forever
    inet6 fe80::146d:4f11:30bf:17a4/64 scope link
        valid_lft forever preferred_lft forever
pi@raspberrypi:~ $
```

Precedidos por un número aparecen todas las interfaces de red con las que cuenta en este momento tu Raspberry Pi. En mi caso `lo`, `eth0` y `wlan0`.

- `lo` es la interfaz de *loopback*, una interfaz virtual que conecta el ordenador consigo mismo. Es muy útil para trabajar con programas de red sin necesidad de usar una red física.
- `eth0` es la interfaz Ethernet cableada (IEEE 802.3). Tanto el modelo B original como los modelos B+, 2B y 3B disponen de una interfaz Ethernet integrada. Los demás modelos pueden incorporarla fácilmente usando un *dongle* USB-Ethernet. Permite hasta Fast Ethernet (velocidades de transferencia de hasta 100 Mbits/s). No es fácil que se actualice a GbE o 10GbE porque internamente la interfaz Ethernet utiliza un puerto USB y de momento los SoC de Broadcom no incorporan nada más que USB 2.0.
- `wlan0` es la interfaz WiFi (*Wireless Local Area Network*). Solo el modelo 3B incorpora interfaz WiFi pero es muy sencillo y muy frecuente añadir una interfaz WiFi USB.

Las redes de comunicaciones hoy en día utilizan en su gran mayoría la familia de protocolos TCP/IP. Se trata de los protocolos que se diseñaron para construir Internet, la red de redes. Hay dos versiones actualmente en uso IPv4 (*Internet Protocol v4*) e IPv6. Aunque son bastante similares hay importantes diferencias que en su mayoría escapan del interés y alcance de este taller. Veremos lo suficiente de IPv4 como para manejarnos en nuestras necesidades de comunicación entre dispositivos y dejaremos IPv6 para futuras ediciones, cuando su grado de adopción sea mayor.

La familia de protocolos TCP/IP está principalmente construida sobre el protocolo IP, que proporciona unas capacidades y garantías básicas. Entre ellas cabe destacar el direccionamiento y el encaminamiento.

Direcciones IPv4

Cada interfaz de red puede tener varias direcciones IP y podemos gestionarlas si así lo deseamos de forma manual. Las direcciones IP funcionan básicamente como códigos postales. Están organizadas de forma jerárquica, lo que facilita la entrega. Por ejemplo, si enviamos una carta al código postal 45005 el cartero sabe que es de Toledo porque empieza por 45. Por tanto llegará en primer término a la central de correos de Toledo, que a su vez la repartirá a sus distintas delegaciones provinciales. En Internet cada central de correos se denomina *router* y las direcciones IP, a diferencia de los códigos postales, identifican plenamente al ordenador destinatario.

Pero ¿cómo llegan los mensajes a un ordenador? Claramente se necesita una interfaz de red. Por tanto las direcciones están asociadas a esa interfaz de red. Puede entenderse como una puerta de una casa. El cartero deja la carta en una puerta, pero nada impide que una casa tenga más de una puerta. Cada puerta tiene asociada una dirección (número) diferente.

Vamos a examinar un poquito la red de mi propia Raspberry Pi y luego veremos cómo configurar la tuya. Nos interesa la última parte de la salida de la orden `ip addr`, que también se puede conseguir así:

```
pi@raspberrypi:~ $ ip addr list wlan0
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:13:ef:71:03:d0 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.38/24 brd 192.168.1.255 scope global wlan0
        valid_lft forever preferred_lft forever
    inet6 fe80::146d:4f11:30bf:17a4/64 scope link
        valid_lft forever preferred_lft forever
pi@raspberrypi:~ $
```

Aparecen tres listas de direcciones: `link/ether` corresponden a lo que se conoce como *direcciones de nivel de enlace*. Es un término técnico muy importante, pero que no veremos en el taller. La etiqueta `inet` corresponde a los datos de IPv4, que son los que usaremos en el taller. La etiqueta `inet6` corresponden a los datos de IPv6 que dejaremos para futuras ampliaciones. Vamos a detenernos en los datos de IPv4.

Campo	Valor	Descripción
Protocolo	<code>inet</code>	Direcciones IPv4.
Dirección	<code>192.168.1.38/24</code>	Dirección de 4 octetos donde los 24 primeros bits son indicativos de la subred a la que pertenece.
Difusión	<code>192.168.1.255</code>	Dirección de difusión a toda la subred.
Ámbito	<code>global</code>	Ámbito de validez de la dirección.
Interfaz	<code>wlan0</code>	Primera interfaz WiFi.

La dirección IP (e.g. `192.168.1.38`) es una secuencia de 4 octetos (números de 0 a 255) habitualmente separados por puntos.

El sufijo `/24` sirve para indicar qué parte de la dirección (en bits contados desde el más significativo) es común para toda la subred y qué parte es específica de cada ordenador de la subred. Con la metáfora del correo equivaldría a determinar qué parte de la dirección es el nombre de la calle y qué parte es el número de casa. En nuestro caso `192.168.1` es común para toda la red y solo el último número es indicativo del ordenador.

El ámbito o *scope* de la dirección indica en qué contextos tiene sentido esa dirección. Las direcciones interesantes tienen alcance global, permiten comunicar al ordenador con cualquiera de los conectados a toda Internet. Otras direcciones tendrán ámbito `host`, como ocurre con la interfaz `lo`. Esas direcciones no están pensadas para comunicar procesos más allá de tu ordenador y sería un error pretender usarla para comunicar dos ordenadores diferentes.

La dirección de *broadcast* o difusión es una dirección especial que se utiliza para enviar mensajes a toda la subred a la que está conectada la interfaz. Se puede construir tomando la parte común de las direcciones de la red (`192.168.1`) y añadiendo unos en todos los bits restantes hasta los 32 que componen una dirección IPv4. En este caso solo quedan ocho bits por completar, que al ponerlos a uno dejarían un 255 en el último octeto.

Vamos a hacer primero pruebas dentro de nuestro propio ordenador y para ello vamos a usar la interfaz de *loopback*.

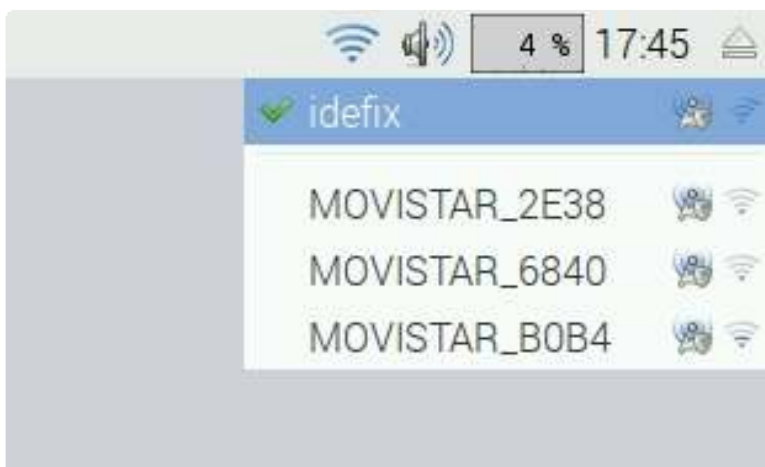
Campo	Valor	Descripción
Protocolo	<code>inet</code>	Direcciones IPv4.
Dirección	<code>127.0.0.1/8</code>	Dirección de 4 octetos donde los 8 primeros bits son indicativos de la subred a la que pertenece.
Ámbito	<code>host</code>	Ámbito de validez de la dirección.
Interfaz	<code>lo</code>	Interfaz de <i>loopback</i> .

Podemos añadir direcciones a la interfaz de *loopback* con:


```
pi@raspberrypi:~ $ sudo ip addr add 192.168.1.39 dev lo
pi@raspberrypi:~ $ ip addr list lo
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet 192.168.1.39/32 scope global lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
pi@raspberrypi:~ $
```

Fíjate que necesitamos utilizar `sudo` para poder añadir direcciones IP, puesto que es una operación de administración.

Así podemos añadir direcciones de forma manual, aunque lo habitual será que éstas se añadan de forma automática. Para ello se emplea un protocolo denominado DHCP (*Dynamic Host Configuration Protocol*). Si conectas un cable Ethernet desde tu router, o desde tu conmutador Ethernet, o uno de los cables de red del laboratorio a la interfaz Ethernet de la Raspberry Pi verás cómo se añade de forma automática una dirección. Lo mismo ocurre con los puntos de acceso WiFi. Por ejemplo pincha en el símbolo WiFi en la parte superior derecha de la pantalla. Verás todas las redes disponibles.



Menú de redes WiFi disponibles.

Selecciona la que corresponde a `eduroam` o a tu punto de acceso WiFi y aparecerá un cuadro de diálogo para que introduzcas los datos de autenticación. Estos datos dependen del tipo de red WiFi, pero es muy similar a la configuración de tu portátil o teléfono móvil, así que seguro que no te resulta complicado.

Cuando estés conectado a la red WiFi verás que aparece una nueva dirección IPv4 y posiblemente otra IPv6 en la interfaz `wlan0`.

Servicio de nombres

¿Y no te extraña que después de todos estos años usando Internet no hayas tenido que usar direcciones IP nunca? Bueno, tal vez en la configuración del punto de acceso WiFi, o para compartir archivos entre móviles, pero poco más. ¿Me equivoco?

El motivo es que este mecanismo de direcciones se simplifica con otro protocolo adicional, el servicio de nombres (DNS, *Domain Name System*). Es un servicio similar a las páginas amarillas de los teléfonos. No tenemos que saber o recordar todos los teléfonos, simplemente tenemos que buscar el nombre en la guía telefónica. El DNS funciona de forma similar. Asocia las direcciones IP a nombres fácilmente recordables, como `www.uclm.es` o `google.com`. Cuando sea necesario utilizaremos el servicio de nombres para obtener la dirección IP correspondiente. Por ejemplo, vamos a averiguar la dirección IP de `www.uclm.es`:

```
pi@raspberrypi:~ $ host www.uclm.es
www.uclm.es has address 161.67.137.169
pi@raspberrypi:~ $
```

Warning Todos los ejemplos de consulta a DNS o conexiones a servidores externos necesitan conexión a Internet. Puedes usar el cable Ethernet de tu puesto de laboratorio.

Puede haber más de una dirección asociada a un nombre. Por ejemplo:

```
pi@raspberrypi:~ $ host twitter.com
twitter.com has address 104.244.42.1
twitter.com has address 104.244.42.65
twitter.com mail is handled by 20 alt1.aspmx.l.google.com.
twitter.com mail is handled by 30 ASPMX3.GOOGLMAIL.com.
twitter.com mail is handled by 20 alt2.aspmx.l.google.com.
twitter.com mail is handled by 10 aspmx.l.google.com.
twitter.com mail is handled by 30 ASPMX2.GOOGLMAIL.com.
pi@raspberrypi:~ $
```

Bueno, aquí vemos que el DNS no solo tiene direcciones IP, pero eso es otra historia y se sale de los objetivos del taller. Lo importante es que observes que hay dos direcciones asociadas a *twitter.com* y cualquiera de ellas se puede usar

Protocolos de transporte

Una red de comunicaciones está para comunicar, así que vamos a ello. En este capítulo solo utilizaremos herramientas de tu Raspberry Pi, sin teclear ni una sola línea de programas. Empezaremos con *netcat*, una maravillosa herramienta para probar programas de red. Abre dos terminales diferentes y ejecuta en el primero lo siguiente:

```
pi@raspberrypi:~ $ nc -l 8888
```

El programa se queda esperando (la opción `-l` significa *listen*, escucha) sin hacer nada. Es lo que se conoce como un servidor. Vete al otro terminal y ejecuta lo siguiente.

```
pi@raspberrypi:~ $ nc 127.0.0.1 8888
Prueba de mensaje
```

Esta vez ejecutamos *netcat* como un cliente conectado a la dirección *127.0.0.1*. Esa es la dirección de la interfaz de *loopback*, así que conectamos con nuestro propio ordenador. Mira en el primer terminal. ¡Magia! Teclea ahora en el primer terminal:

```
pi@raspberrypi:~ $ nc -l 8888
Prueba de mensaje
Otra prueba de mensaje
```

¿Se ve? Tienes un canal bidireccional de datos, puedes enviar y recibir indistintamente. Para salir pulsa la tecla *Ctrl* y sin soltarla pulsa la letra *C*.

El número 8888 es lo que se conoce como un número de puerto. Si seguimos con la metáfora de correos equivaldría al buzón dentro de una casa de pisos. Se reserva un puerto para cada servicio. Los puertos por debajo de 1024 son privilegiados en el sentido de que se utilizan para servicios del sistema y por tanto un usuario normal no puede atender peticiones en esos puertos. Veamos un ejemplo, el puerto 80 es el puerto del servidor web. Ejecuta esto:

```
pi@raspberrypi:~ $ sudo ip addr add 161.67.137.169 dev lo
pi@raspberrypi:~ $ sudo nc -l 80 << EOF
HTTP/1.0 200 OK

<html><body><h1>You fool!</h1></body></html>
EOF
```

Ahora ejecuta el navegador web e intenta ver la página *www.uclm.es*. ¿Sorprendido?

Hemos añadido una dirección IP a la interfaz de *loopback* que coincide con la de *www.uclm.es*. Cuando introducimos este sitio en el navegador éste utiliza el DNS para obtener la dirección IP correspondiente y envía un mensaje de petición a esa dirección IP. Como se trata de una dirección IP conocida el mensaje ni siquiera sale del ordenador, se lo queda la interfaz de *loopback*. En todas las direcciones del ordenador tenemos a *netcat* escuchando en el puerto 80, así que ese mensaje de petición le llega a *netcat*. Lo que hay entre `<< EOF` y `EOF` no es más que una forma de indicar que eso es lo que debe escribir por entrada estándar. Y lo que se escribe por entrada estándar *netcat* se encarga de mandarlo al otro extremo de la comunicación. ¡Voilà! Un servidor web de pobres.

Este ejemplo te habrá dejado claro por qué las direcciones IP solo las puede poner el administrador del sistema. En caso contrario sería trivial hacer ataques de *man-in-the-middle* contra los usuarios de ese ordenador. Yo creo que ya puedes borrar esa dirección de la interfaz de *loopback*.

```
pi@raspberrypi:~ $ sudo ip addr del 161.67.137.169 dev lo
pi@raspberrypi:~ $ _
```

Ya sabes comunicar datos, pero no solo entre dos terminales del mismo ordenador. Si estás conectado a la red prueba a conectar los dos *netcat* ejecutando cada uno en distinto ordenador. ¡Ya sabes comunicar datos con TCP/IP!

Transmission Control Protocol

Internet es mucho más hostil de lo que puedes apreciar con una prueba en tu propia Raspberry Pi. Las comunicaciones entre ordenadores que distan más de diez mil kilómetros involucran a decenas de dispositivos intermedios. Hay una probabilidad nada despreciable de que algo no vaya bien. Pérdidas de mensajes, o incluso reorganizaciones de la red son normales durante el mismo proceso de comunicación. ¿Qué debes hacer entonces? La respuesta es a la vez simple y reconfortante: nada. Lo que se puede hacer ya lo hace el protocolo subyacente, TCP (*Transmission Control Protocol*). Cuando usabas *netcat* estabas usando TCP sin saberlo. Se trata de un *protocolo de transporte* construido sobre IP. Este protocolo proporciona mecanismos de control de flujo, fragmentación, reensamblado, integridad de datos, reordenación y retransmisión para que la experiencia de comunicación sea lo más parecida posible a la ideal, incluso aunque las cosas no vayan bien. Es tan importante que a toda la familia de protocolos de Internet se le llama TCP/IP, aunque TCP e IP son solo dos protocolos de la familia.

No cabe duda de que ha tenido éxito. La gran mayoría de los servicios a gran escala de Internet se construyen sobre TCP: la web, el correo electrónico, los sistemas de mensajería instantánea, el servicio de directorio, y prácticamente todo lo que requiera cifrado de extremo a extremo.

Sin embargo para proporcionar garantías de entrega TCP tiene que utilizar una serie de mecanismos que consumen tiempo, y es un tiempo que no es fácilmente controlable. Se dice que TCP *introduce latencia no acotada*. Es decir, no es posible cuantificar exactamente el tiempo máximo que tardará en llegar un mensaje a su destino, ni siquiera poner un límite superior.

Piensa en sistemas que requieran una respuesta en un tiempo acotado. Es lo que se conoce como *sistemas de tiempo real*. Hay de todo tipo, desde sistemas de control de maquinaria, hasta sistemas multimedia.

- *Sistemas de tiempo real crítico (Hard Real-Time Systems)*. Un sistema de control de un robot, un avión o un helicóptero no se puede plantear la retransmisión indefinida de los mensajes. Si hay un problema debe tratarse a tiempo para garantizar en todo momento la seguridad de los usuarios y la propia integridad física del aparato. En ocasiones los tiempos de respuesta no pueden superar un puñado de microsegundos y si no responde a tiempo el resultado puede ser fatal. Este tipo de sistemas no se puede tratar con GNU/Linux tal y como lo hemos visto hasta ahora. No descartamos un futuro taller para este tipo de sistemas, pero desde luego se sale de los objetivos del curso actual.
- *Sistemas de tiempo real blando (Soft Real-Time Systems)*. En otros casos la respuesta en un tiempo acotado es deseable, pero no tiene consecuencias catastróficas si no se cumple. Es el caso de los sistemas multimedia. Cuando se realizan comunicaciones de voz y vídeo es deseable que la tasa de llegadas sea más o menos constante y sobre todo que no se acumulen retrasos de más de la duración del *buffer*. Si no se cumple veremos cortes en el vídeo y molestos clicks metálicos en el audio. Este tipo de sistemas se pueden tratar sin problemas con GNU/Linux pero TCP no suele ser la mejor opción posible.

Unreliable Datagram Protocol

Todos los sistemas de tiempo real tienen un requisito común: necesitan reducir la latencia al mínimo posible. La latencia es el tiempo que pasa desde que se envía un mensaje hasta que se entrega en el otro extremo.

Para tratar estos problemas existe un protocolo complementario de TCP, el *Unreliable Datagram Protocol* (UDP). En UDP no hay control de flujo, no hay retransmisiones, no hay reordenación, no hay nada más que encaminamiento y garantía de integridad. Osea, sabemos cómo llevar los mensajes a su destino y si llegan seguro que son los que se enviaron. Eso es todo. Pueden llegar en orden distinto al de envío, pueden perderse por el camino, pueden incluso duplicarse. Y el emisor nunca tendrá ningún tipo de realimentación sobre si el mensaje ha sido recibido correctamente o no. Poco, ¿verdad?

La parte positiva es que la latencia, especialmente en redes con pérdidas o congestionadas, es sensiblemente inferior. Así que se usa para difusión de vídeo y audio. Por su simplicidad también se suele usar para comunicar con dispositivos muy pequeños (microcontroladores de 8 bits, FPGAs, etc.). Todas las garantías necesarias quedan como responsabilidad de la

aplicación. Por ejemplo, si necesita confirmaciones debe enviar mensajes UDP solicitando esas confirmaciones.

Bueno, suficiente teoría, manos a la obra. Prepara los dos terminales como en el caso de la comunicación TCP anterior y ejecuta lo siguiente en el primero.

```
pi@raspberrypi:~ $ nc -u -l 8888
```

Este es el servidor UDP. Servidor es el nombre del rol pasivo en la comunicación, el que espera. Eso no significa que no hable, pero no habla hasta que alguien (un cliente) inicia una conversación.

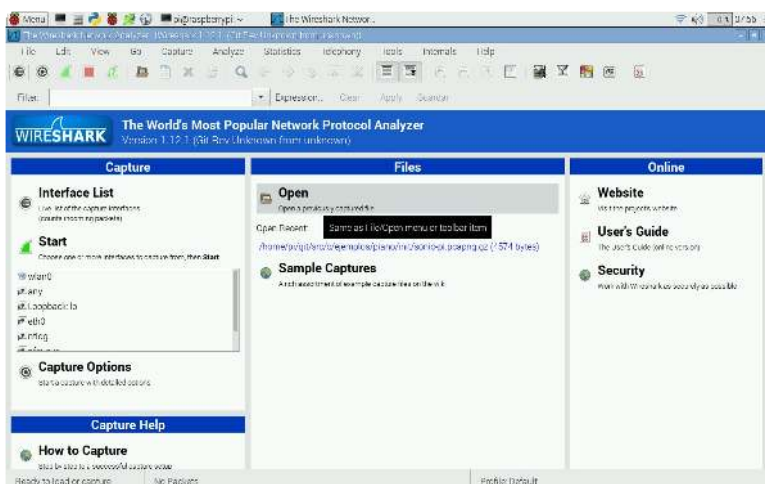
En el otro terminal ejecuta el cliente:

```
pi@raspberrypi:~ $ nc -u 127.0.0.1 8888
```

Prueba a transitar datos en los dos sentidos. Como ves funciona de forma muy similar y no podrás apreciar la diferencia si no usas equipos relativamente distantes y redes algo congestionadas. Para los efectos del taller son prácticamente equivalentes, y nos decantaremos por uno u otro según los requisitos de la aplicación.

Wireshark

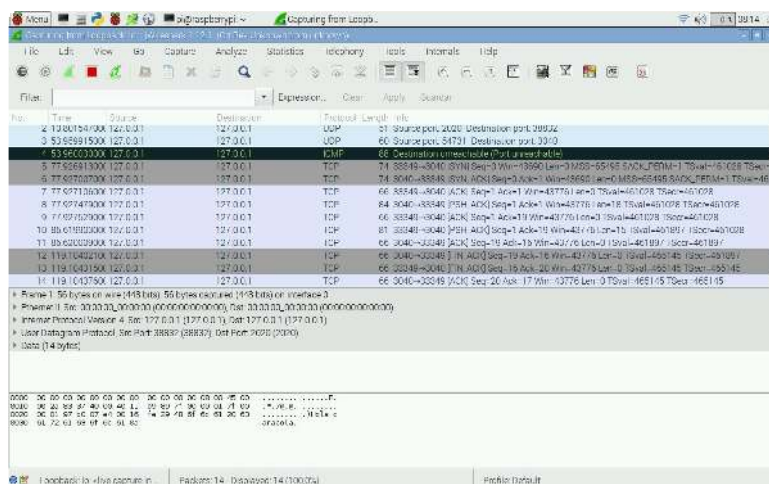
Cuando se utilizan comunicaciones en red es muy frecuente que aparezcan problemas de todo tipo. ¿Cómo los diagnosticamos? La forma más sencilla es utilizar una herramienta de captura y análisis de tráfico. En tu Raspberry Pi ya tienes instalada una de las mejores herramientas disponibles: *Wireshark*.



Pantalla inicial de Wireshark.

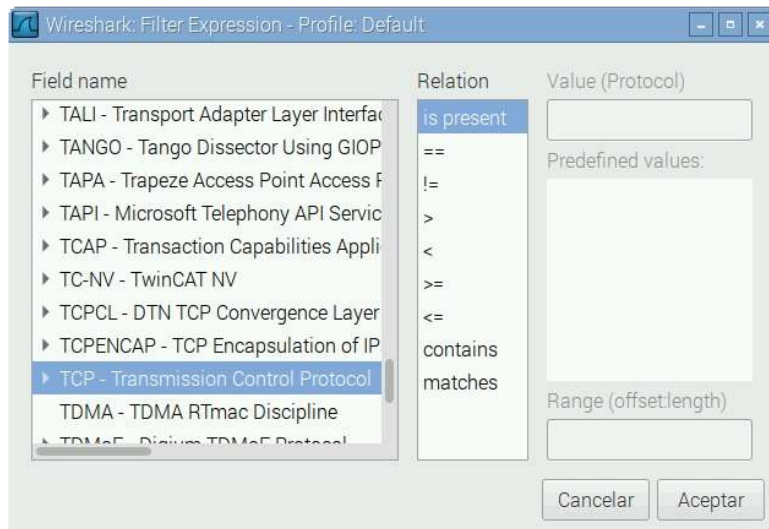
Al arrancar la aplicación muestra algo parecido a la figura. Aparece una lista con las interfaces de red que ya conocemos y alguna más que se escapa de lo que se pretende con este taller. Selecciona la interfaz *loopback: lo* y pulsa en *Start*.

Ahora vuelve a repetir el ejemplo de comunicación TCP y UDP que hemos hecho en este capítulo. Es interesante incluso equivocarse a propósito. Prueba por ejemplo a ejecutar el cliente UDP sin que haya servidor al otro extremo. El resultado será algo así:



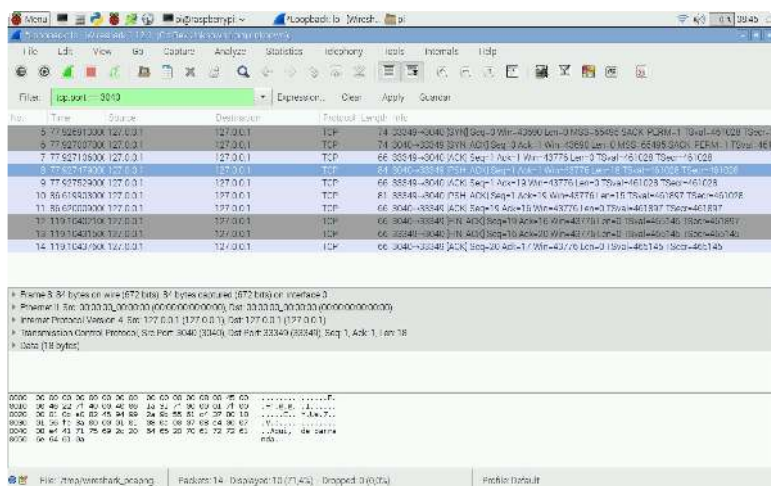
Captura de mensajes de la interfaz "loopback".

Lo normal en una red es ver centenares de paquetes y es relativamente difícil encontrar lo que nosotros queremos examinar. Por eso *Wireshark* incorpora características de filtrado muy avanzadas. Para la captura pulsando sobre el botón del cuadrado rojo y vamos a filtrar los paquetes que nos interesan. Por ejemplo, vamos a ver los paquetes TCP destinados al puerto que hayas usado. Para ello pulsa en *Expression...* Tendrás un diálogo como el de la figura en el que puedes escribir TCP para encontrar todos los campos de TCP.



Edición de filtros de la captura.

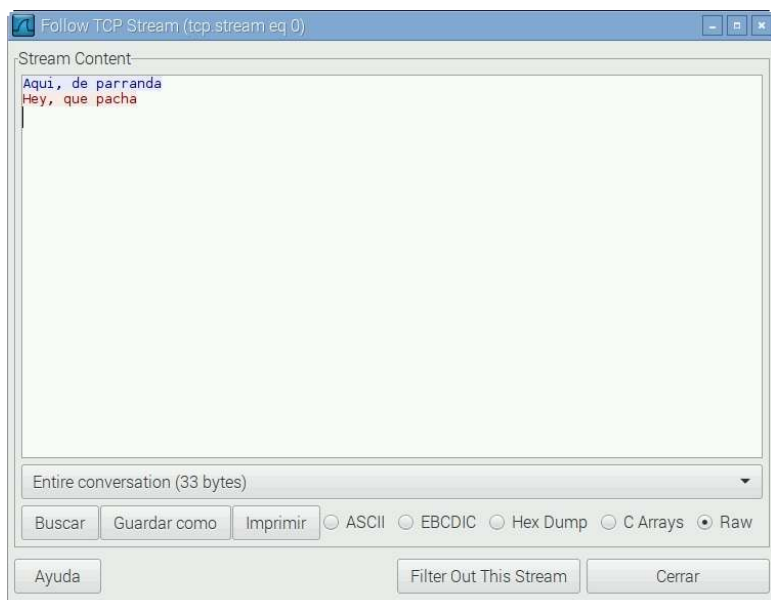
Despliega todos los parámetros del protocolo TCP y selecciona `tcp.port`. En la columna *Relation* elige `==` y en la columna *Value* elige el número de puerto que hayas usado. Para terminar pulsa *Aceptar*. Ahora verás en la casilla *Filter* la expresión que has seleccionado, algo del estilo a `tcp.port == 3040`. Pero todavía no la hemos aplicado. Pulsa al botón *Apply*. Ya solo aparece la conversación que nos interesa. Como ves las cosas son más complicadas de lo que parecen:



Conversación TCP con un solo mensaje (Len distinto de 0).

Los tres primeros mensajes son el establecimiento de conexión (*triple handshake*) y hasta el cuarto mensaje no vemos lo que nosotros hemos enviado. En la ventana centras aparecen diseccionados los distintos elementos de cada mensaje capturado.

Otra opción muy útil es el seguimiento del flujo completo. Pulsa el botón derecho sobre cualquiera de los mensajes en la parte superior de la pantalla y selecciona la opción *Follow TCP Stream* en el menú contextual que te aparece.



Seguimiento del flujo TCP.

Aparece toda la secuencia de intercambio de datos sin toda la parafernalia de TCP. Solo los datos. En azul en un sentido y en rojo en el otro. Además podemos seleccionar solo el tráfico en uno de los sentidos. Esto es extraordinariamente útil cuando queremos replicar el comportamiento de otros programas.

Programando en red

La historia de las redes de comunicaciones entre ordenadores está muy relacionada con la historia de Unix y por tanto indirectamente con GNU/Linux. En la Universidad de California en Berkeley se generalizó la idea original de Unix, en la que todo es un archivo, a las redes. Las comunicaciones se programan de forma muy similar al uso de archivos. Estos archivos especiales se denominan *sockets*.

La interfaz de programación basada en *sockets* (API socket) sigue siendo la dominante hoy en día para tratar con redes de comunicaciones y es la que veremos en el capítulo correspondiente.

De todas formas conviene dar unas pequeñas nociones de los fundamentos generales antes de escribir código. En la introducción a GNU, en el capítulo 1 de este manual, ya hablamos de los *descriptores de archivo*. Son números que representan a los archivos desde el punto de vista del sistema operativo. Básicamente proporcionan una interfaz basada

en cuatro operaciones básicas, que corresponden a *llamadas al sistema* (servicios del sistema operativo):

- La llamada *open* abre un archivo. Busca el archivo en el sistema de archivos usando un nombre jerárquico que se denomina *ruta del archivo* y asigna un nuevo descriptor de archivo. A partir de este momento nos podemos olvidar de la ruta. El sistema operativo solo necesita el descriptor.
- La llamada *close* cierra el archivo. Con esto el sistema operativo termina todas las operaciones en curso que afectan al archivo y libera su descriptor para que pueda ser reutilizado con otro archivo.
- La llamada *write* escribe en el archivo un conjunto de octetos y la llamada *read* lee del archivo un conjunto de octetos.

Esta interfaz de programación se utiliza en GNU para multitud de operaciones, las operaciones con archivos del disco, la escritura de mensajes en el terminal, la lectura de datos del teclado, la lectura de eventos del ratón, etc. Parece lógico que se extienda también a la programación de comunicaciones en red y eso es lo que hace la interfaz *socket*.

Una conexión de red no tiene ningún elemento en el disco que la represente. Por tanto no queda más remedio que sustituir la llamada *open* por otra equivalente que proporcione la información necesaria para la comunicación. Esa llamada se llama *socket* y es la que da nombre a la interfaz de programación. La llamada *socket* asigna un descriptor de archivo a un canal de comunicación. Pero el canal de comunicación no basta, necesitamos proporcionar los datos acerca del destinatario o el origen de cada mensaje (direcciones IP que utiliza, puertos TCP o UDP, etc.). Esto se hace con una nueva llamada *connect* para el lado del cliente o *bind* para el lado del servidor. Cuando el *socket* está conectado ya funciona como un descriptor de archivo normal.

A partir de ese momento todas las llamadas de descriptors de archivo son también válidas (*read*, *write* y *close* son también operaciones que se pueden realizar sobre *sockets* una vez que están conectados).

Las comunicaciones UDP no necesitan más llamadas, esto es todo. Es tan similar que en algunos sistemas operativos como Plan9 o 2k los *sockets* se crean con llamadas a *open* usando una ruta de archivo especial. A pesar de todo GNU proporciona otras llamadas *sendto* o *recvfrom* que no hacen sino combinar la llamada *write* con la llamada *connect* y la llamada *read* con la llamada *bind*. No las usaremos en absoluto.

Sin embargo TCP es mas complejo. Para proporcionar las garantías adicionales necesita que se implemente el concepto de *conexión* como el establecimiento de un nuevo canal único entre los dos procesos que se comunican. En el lado del cliente es sencillo, basta con que *connect* haga este trabajo. La conexión en este caso no es simplemente asignar la dirección destino, sino reservar una serie de recursos para la comunicación con el destino.

El lado del servidor TCP es el más complejo. Cada cliente que se conecte con el servidor tiene que disponer de su propia conexión. Con *socket* y *bind* podemos crear un socket que atiende mensajes destinados a una dirección concreta y a un puerto concreto, pero solo uno. La solución pasa por una llamada más, *accept*, que crea otro nuevo socket en cada conexión desde un proceso cliente. La API se completa con una llamada *listen* que es preciso invocar para configurar cuantas conexiones casi simultáneas es posible atender.

Aunque hay muchos más detalles que ni hemos mencionado creo que puede servir para entender cómo se programan las comunicaciones. Estas tablas pueden servir para entender la secuencia de operaciones que se debe realizar.

Servidor UDP	Cliente UDP	Descripción
<code>socket</code>	<code>socket</code>	Crea canal de comunicación.
<code>bind</code>	<code>connect</code>	Configura direcciones origen/destino.
<code>read</code>	<code>read</code>	Recibe datos del otro extremo.
<code>write</code>	<code>write</code>	Envía datos al otro extremo.
<code>close</code>	<code>close</code>	Cierra el canal de comunicación.

Servidor TCP	Cliente TCP	Descripción
socket	socket	Crea canal de comunicación.
bind	connect	Configura direcciones origen/destino.
listen		Configura el número de conexiones simultáneas.
accept		Crea socket esclavo para atender conexión.
read	read	Recibe datos del otro extremo.
write	write	Envía datos al otro extremo.
close	close	Cierra el canal de comunicación.

Desarrollo en C

Por primera vez el taller se realizará en versión multi-lenguaje (C y Python). Esperamos de esta forma acomodarnos a un número mayor de alumnos. En la actualidad usamos Python como primer lenguaje de la titulación (primer semestre de primer curso, asignatura *Informática*) pero hasta este curso hemos utilizado C. Por tanto hay un grupo de estudiantes que se sienten más cómodos con C y otro que se siente más cómodo con Python.

Este manual trata de dividir el contenido de forma que no tengas que leer todo si lo que te interesa es uno solo de estos lenguajes. En este capítulo describiremos el conjunto de herramientas que vamos a usar para hacer programas en C y un ejemplo sencillo de su uso.


El primer programa

Es tradición empezar por un primer programa que simplemente escribe el mensaje `Hola, Mundo` por la pantalla. Se remonta a los [orígenes de BCPL](#), el precursor de C.

Arranca un terminal de órdenes y crea una carpeta para meter los ejemplos de este capítulo. Trabajaremos en esa carpeta. Por ejemplo:

```
pi@raspberrypi:~ $ mkdir test
pi@raspberrypi:~ $ cd test
pi@raspberrypi:~/test $ _
```

Utiliza un editor de texto para escribir el siguiente programa en un archivo de nombre `hola.c`. Como editor de textos

puede usarse *leafpad*, simplemente pulsando sobre el icono  o desde la línea de órdenes con `leafpad hola.c`.

```
#include <stdio.h>

int main() {
    puts("Hola, Mundo");
    return 0;
}
```

Graba el archivo en la carpeta que habías creado y vuelve al terminal:

```
pi@raspberrypi:~/test $ make hola
cc      hola.c      -o hola
pi@raspberrypi:~/test $ _
```

El programa en C ha sido compilado por GNU make (`make`) que a su vez ha ejecutado el compilador de C (`cc`) para generar el ejecutable `hola`. Fíjate que en GNU los ejecutables no tienen una extensión distintiva.

Ahora podemos ejecutarlo:

```
pi@raspberrypi:~/test $ ./hola
Hola, Mundo
pi@raspberrypi:~/test $ _
```

Fíjate que hemos escrito un punto y una barra delante para indicar dónde se encuentra el archivo. Es una ruta relativa, de las que ya hemos hablado.

Te preguntarás por qué tenemos que indicar la ruta para este ejecutable y no para `leafpad` por ejemplo. El sistema operativo encuentra los ejecutables o bien porque el usuario le dice explícitamente dónde están o bien porque están en una serie de carpetas que se conocen como las *rutas del sistema*. Evidentemente la carpeta actual no está en las rutas del sistema, porque la acabas de crear, así que nuestra única opción es indicar la ruta.

Warning Es posible que leas textos que te recomiendan añadir la carpeta `.` (la carpeta de trabajo) a las rutas del sistema. No lo hagas, es una mala idea en general pero especialmente por motivos de seguridad.

El compilador de C

En GNU/Linux hay varios compiladores de C. El más utilizado es el del proyecto GNU (*GNU Compiler Collection*) que se invoca con la orden `gcc` o simplemente `cc`. Normalmente no compilaremos a mano los archivos C, sino que lo haremos a través de una herramienta de construcción, concretamente *GNU make*. Sin embargo es necesario conocer cómo funciona y qué opciones tiene el compilador.

GNU Compiler Collection (GCC) es un conjunto de compiladores de muy diversos lenguajes (C, C++, Objective-C, Java, D, Pascal, Ada, etc.) y de muy alta calidad. Se trata de una herramienta muy completa y compleja, y no es objeto de este manual presentarla en profundidad. Baste decir que se ha convertido en un estándar de facto en la industria, que se utiliza incluso por los diseñadores de procesadores para probar los nuevos diseños antes incluso de su producción comercial.

Las opciones más comunes son:

Opción	Significado
<code>-c</code>	Compila o ensambla, pero no enlaza, obteniéndose un archivo en código objeto con extensión <code>.o</code>
<code>-Wall</code>	Emite todos los avisos que el compilador pueda generar
<code>-E</code>	Realiza únicamente el preprocesamiento, enviando el resultado a la salida estándar
<code>-ggdb</code>	Incluye información de depuración
<code>-Iruta</code>	Especifica la <i>ruta</i> de una carpeta donde buscar archivos de cabecera
<code>-Lruta</code>	Especifica la <i>ruta</i> de una carpeta donde buscar bibliotecas
<code>-lXXX</code>	Enlaza con la biblioteca de nombre <code>libXXX.a</code>
<code>-o nombre</code>	Indica el nombre del archivo ejecutable
<code>-v</code>	Muestra las fases por las que va pasando el compilador
<code>-w</code>	Suprime los mensajes de aviso (<i>warnings</i>)
<code>-S</code>	Preprocesa y compila, pero no ensambla ni enlaza

El compilador de C puede usarse como compilador, como montador o como ambas cosas. Por ejemplo, volvamos a nuestro programa `hola.c` y escribe en el terminal lo siguiente:

```
pi@raspberrypi:~/test $ gcc -c hola.c
pi@raspberrypi:~/test $ ls
hola hola.c hola.o
pi@raspberrypi:~/test $
```

Ahora tenemos un archivo `hola.o`. Se trata de un *archivo objeto*, con las instrucciones máquina correspondientes al archivo `hola.c` pero sin la estructura de un ejecutable. Varios archivos objeto se pueden combinar para generar un único ejecutable usando el *montador*. En GNU se puede usar el propio compilador de C para montar el ejecutable:

```
pi@raspberrypi:~/test $ gcc -o hola hola.o
pi@raspberrypi:~/test $
```

Cuando no usamos la opción `-c` el compilador se comporta como montador y si es necesario como compilador simultáneamente. Si no especificamos un nombre de ejecutable con la opción `-o` el compilador generará uno con nombre `a.out`. Esto es así por razones históricas, por lo que es evidente que normalmente debemos especificar la opción `-o`.

Múltiples archivos fuente

Un ejecutable puede componerse a partir de varios archivos fuente. Por ejemplo, divide el programa en un archivo `hola.c` que tiene el programa principal y utiliza una función `decir_hola` que está definida en otro archivo de nombre `f.c`:

```
#include <stdio.h>

void decir_hola() {
    puts("Hola, Mundo");
}
```

Ahora el archivo `hola.c` sería algo así:

```
void decir_hola(void);

int main() {
    decir_hola();
    return 0;
}
```

Para generar el ejecutable habría que compilar ambos archivos y luego montarlos:

```
pi@raspberrypi:~/test $ gcc -c hola.c f.c
pi@raspberrypi:~/test $ gcc -o hola hola.o f.o
pi@raspberrypi:~/test $ █
```

Vamos a cambiar el programa para que salude y se despidas. Además de la función `decir_hola` (ahora en el archivo `f_hola.c`) tendremos otra función `decir_adios` (en el archivo `f_adios.c`).

```
pi@raspberrypi:~/test $ mv f.c f_hola.c
pi@raspberrypi:~/test $ cp f_hola.c f_adios.c
pi@raspberrypi:~/test $ leafpad f_adios.c
```

Ahora cambia el archivo `f_adios.c` para que se despidas:

```
#include <stdio.h>

void decir_adios() {
    puts("Adios, Mundo");
}
```

Y el programa principal `hola.c` para que llame a las dos funciones:

```
void decir_hola(void);
void decir_adios(void);

int main() {
    decir_hola();
    decir_adios();
    return 0;
}
```

Esto se va pareciendo más a un programa de verdad. El programa principal llama a varias funciones que están repartidas por otros archivos. Pero no es habitual que se ponga la declaración de las funciones directamente en el archivo principal. Es mejor poner esas declaraciones en un archivo de cabecera que se incluye cuando se necesita.

Por ejemplo, edita un nuevo archivo `saludar.h` con las declaraciones de las dos funciones:

```
#ifndef SALUDAR_H
#define SALUDAR_H
void decir_hola(void);
void decir_adios(void);
#endif
```

Y úsalo en `hola.c`:

```
#include "saludar.h"

int main() {
    decir_hola();
    decir_adios();
    return 0;
}
```

Ahora podemos compilar todo:

```
pi@raspberrypi:~/test $ gcc -c hola.c f_hola.c f_adios.c
pi@raspberrypi:~/test $ gcc -o hola hola.o f_hola.o f_adios.o
pi@raspberrypi:~/test $ █
```

Fíjate que el archivo de cabecera no es necesario compilarlo porque no tiene nada más que declaraciones y se incluye ahí donde es necesario.

Bibliotecas de programas

Cuando los programas van creciendo empieza a ser necesario algún mecanismo de organización más flexible. Por ejemplo, podemos agrupar varios de los archivos objeto en una biblioteca y luego montar el ejecutable con la biblioteca. Por ejemplo, vamos a meter los archivos `f_hola.o` y `f_adios.o` en una biblioteca `libsaludar.a` y después construimos el ejecutable con esta biblioteca. Para crear la biblioteca usaremos el programa `ar` (*archiver*):

```
pi@raspberrypi:~/test $ ar rcs libsaludar.a f_hola.o f_adios.o
pi@raspberrypi:~/test $ gcc -L. -o hola hola.o -lsaludar
pi@raspberrypi:~/test $ █
```

El programa `ar` es en cierta forma similar a los archivadores que se utilizan para crear archivos comprimidos (WinZIP, WinRAR, 7zip, PeaZIP, etc.) salvo que en este caso no se comprime. En GNU la compresión y el archivado son procesos separados, de manera que el usuario pueda elegir cómo archiva y cómo comprime de forma independiente.

En el mismo archivo `libsaludar.a` podemos meter cualquier número de archivos objeto con cualquier número de funciones. Sin embargo a la hora de construir el ejecutable con `gcc` las cosas no cambian aunque el número de archivos y de funciones crezca.

Las bibliotecas suelen hacerse en una carpeta independiente. Por ejemplo:

```
pi@raspberrypi:~/test $ mkdir saludo
pi@raspberrypi:~/test $ mv *.h f_* saludo
pi@raspberrypi:~/test $ cd saludo
pi@raspberrypi:~/test/saludo $ gcc -c *.c
pi@raspberrypi:~/test/saludo $ ar rcs libsaludar.a *.o
pi@raspberrypi:~/test/saludo $ cd ..
pi@raspberrypi:~/test $ █
```

Ahora la biblioteca está en la subcarpeta `saludo` y el programa principal en la carpeta padre. Para compilar `hola.c` tenemos que decirle al compilador dónde pueden estar los archivos de cabecera y al montar `hola` debemos indicar la nueva carpeta donde buscar bibliotecas:

```
pi@raspberrypi:~/test $ gcc -c -Isaludo hola.c
pi@raspberrypi:~/test $ gcc -Lsaludo -o hola hola.o -lsaludar
pi@raspberrypi:~/test $ █
```

La herramienta de construcción *GNU make*

Como hemos visto, en cuanto el programa empieza a crecer un poco el proceso de construcción del ejecutable puede ser realmente tedioso. El programa `make` que ya usamos en el primer ejemplo nos permite automatizar la construcción.

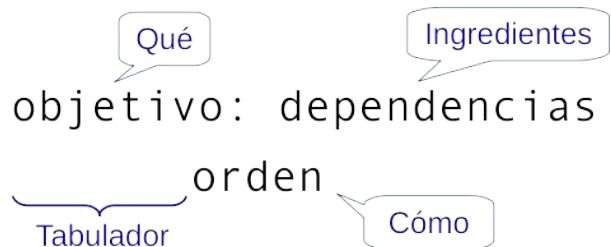
Salvo en los casos triviales `make` necesita de un archivo de configuración que le indique cómo construir el ejecutable. Debe llamarse `makefile`, `Makefile` o `GNUmakefile`. Básicamente se compone de recetas en las que se le explica a `make` cómo construir un archivo a partir de un conjunto de archivos especificado. Así, en nuestro último ejemplo podemos hacer un `makefile` dentro de la subcarpeta `saludo` como éste:

```
libsaludar.a: f_hola.o f_adios.o
ar rcs $@ $^
```

Esta regla se lee así: *para construir `libsaludar.a` debes construir primero `f_hola.o` y `f_adios.o` y entonces ejecutar `ar rcs libsaludar.o f_hola.o f_adios.o`*. Date cuenta de que usamos la abreviatura `$@` para representar el objetivo de la regla (`libsaludar.a`) y `$^` para representar las dependencias de la regla (todo lo que sigue a los dos puntos).

¿Y cómo le decimos como hacer los archivos objeto `f_hola.o` y `f_adios.o`? No es necesario. *GNU make* sabe cómo hacerlos a partir de archivos C con el mismo nombre. Internamente tiene ya definida una receta como esta:

```
%.o: %.c
$(CC) $(CFLAGS) -c $<
```



Que se lee *para construir un archivo objeto a partir de un archivo C del mismo nombre debes ejecutar*

Anatomía de una receta de make.

`$(CC) $(CFLAGS) -c $<` Donde el símbolo `$<` corresponde a una variable especial de *GNU make*.

Variable	Significado
<code>\$@</code>	Objetivo de la receta
<code>\$^</code>	Dependencias de la receta
<code>\$<</code>	Primera dependencia de la receta

Las recetas incluidas ya en *GNU make* hacen que prácticamente no sea necesario definir nuevas reglas salvo en casos muy sencillos.

Lo interesante de esto es que las reglas de *make* utilizan una amplia variedad de variables que ya tienen un valor por defecto, pero que podemos cambiar para ajustar el funcionamiento. Veamos unas cuantas:

Variable	Contenido
CC	Compilador de C (por defecto <code>cc</code>)
CFLAGS	Banderas del compilador (opciones que se le pasan al compilador)
LDLFLAGS	Banderas del montador (opciones que se le pasan al montador)
LDLIBS	Bibliotecas que deben incluirse en el ejecutable
RM	Programa que borra archivos (por defecto <code>rm</code>)

Por tanto en la carpeta del archivo `hola.c` podemos poner un `makefile` como éste:

```
CFLAGS=-Isaludo
LDLFLAGS=-lsaludar
LDLIBS=-lsaludo

hola: hola.o
```

Es también habitual poner una regla para limpiar los archivos intermedios generados:

```
clean:
$(RM) *.o *~ hola
```

Ahora basta ejecutar `make clean` para borrar los archivos generados y dejar solo el código que hemos escrito.

Un aspecto interesante del uso de `make` es que solo genera lo necesario, y se aplican las reglas afectadas por archivos que han cambiado. Es decir, si se dispone de un buen `makefile` basta con ejecutar `make` para que se compile todo lo necesario y solo lo necesario en cualquier momento.

Si el `makefile` no tiene todas las dependencias *GNU make* no sabrá si tiene que reconstruir algún archivo. Por ejemplo, en este pequeño ejemplo no hemos añadido dependencias de `hola.o` con `saludo/saludar.h`. Lo correcto habría sido añadir una regla en la que solo se indica la dependencia:

```
hola.o: hola.c saludo/saludar.h
```

Si no quieres complicar los `makefile` basta tenerlo presente y cuando se modifique algo que no está en las dependencias podemos forzar la compilación con:

```
pi@raspberrypi:~/test $ make -C saludo -B
pi@raspberrypi:~/test $ make -B
```

La primera orden fuerza la construcción de la biblioteca y la segunda fuerza la construcción del ejecutable. Éstas son las opciones más frecuentes de `make`:

Opción	Significado
<code>-B</code>	Fuerza la reconstrucción de todo
<code>-c carpeta</code>	Cambia primero a la carpeta indicada y luego ejecuta <i>make</i>
<code>-d</code>	Imprime información de depuración
<code>-n</code>	Solo imprime las órdenes que ejecutaría, pero no las ejecuta
<code>var=valor</code>	Asigna <code>valor</code> a la variable <code>var</code> para esta ejecución de <i>make</i>

Cae fuera del ámbito de este curso explicar con más detalle la sintaxis que utiliza el archivo `makefile`. En su lugar, utilizaremos `make` con archivos `makefile` proporcionados en plantillas de aplicaciones. Consulta la página de manual y la documentación en línea [Citation not found] para ayudarte a comprenderlos.

Depuración de programas con *GNU debugger*

La construcción de programas correctos requiere de la estrecha cooperación de dos técnicas muy sencillas pero muy importantes: la prueba y la depuración.

La prueba consiste en la elaboración de pequeños programas que demuestran que nuestras funciones hacen lo que se supone que deben hacer. No hay nada que sustituya a esto y es esencial en todo proceso de desarrollo de software. Trataremos algo este tema más adelante.

La otra técnica que complementa a la prueba es la depuración, que consisten en la eliminación de errores (*bugs*) previamente detectados en las pruebas. La depuración es un proceso relativamente sencillo en la forma y totalmente sistemático, pero sorprendentemente difícil en la práctica.

No se necesita ninguna herramienta especial para depurar. Se puede emplear un conjunto de sentencias `printf` cuidadosamente elegidas o cualquier otra forma de examinar el contenido de la memoria. *GNU debugger* (abreviado `gdb`) no es por tanto estrictamente necesaria pero puede ahorrarte una enorme cantidad de tiempo.

Las herramientas de depuración permiten detener la ejecución del programa en puntos específicos, por ejemplo, justo antes de la ocurrencia de un error en el que se sabe que la ejecución termina inesperadamente. La detención del programa permite explorar en ese punto la memoria, incluso conocer el valor de las variables y registros del procesador e ir avanzando por la ejecución hasta encontrar el punto exacto en el que ocurre la disfunción del programa.

El proceso de depuración

Hablemos primero un poquito acerca del propio proceso de eliminación de errores. Es importante porque la experiencia nos dice que es contra-intuitivo, tendemos de forma natural a saltar pasos esenciales.

Los errores en los programas se pueden clasificar de varias formas. Es muy habitual clasificarlos según su visibilidad y su persistencia:

	Visible	Oculto
Persistente	Ideal	Desconocido
Transitorio	Difícil	Muy difícil

Un error *visible* es aquél que se ha detectado porque el programa no hace lo que debe. Por el contrario un error *oculto* es aquél que no se ha detectado aunque existe, ya sea porque el código al que afecta se ejecuta muy raramente o porque hay otro error que lo enmascara. Los errores visibles se suelen detectar en las pruebas y no suelen llegar al código en producción. Sin embargo los ocultos llegan al código de producción y pueden tener consecuencias desastrosas. *Ariane 5* o *Therac 25* son algunos ejemplos que todos deberíamos recordar. Busca en Google si no los conoces.

Por otro lado también pueden ser clasificados en errores *persistentes* cuando los errores se manifiestan en todas las ejecuciones del programa o *transitorios* cuando solo se manifiesta en algunas ejecuciones o en momentos impredecibles.

Obviamente los ideales para su eliminación son los errores *visibles* y *persistentes* y tenemos que hacer todo lo posible para que los posibles errores sean de este tipo. Para eso se utilizan técnicas de programación defensiva y herramientas de depuración de memoria como `valgrind`. Cae fuera del ámbito de este curso hablar de estas últimas, pero te recomendamos que las pruebes si encuentras con un error elusivo.

Habitualmente los errores se detectan en las pruebas, e incluso aunque no fuera así deberíamos hacer un programa pequeño de prueba que reproduzca el error. Por tanto podemos asumir que existe un programa relativamente corto que no hace lo que esperamos.

La clave de la depuración consiste en aplicar de forma *consistente* el siguiente proceso:

- Estudia los datos. Mira qué pruebas fallan y cuáles tienen éxito.
- Elabora una *hipótesis* consistente con los datos.
- Diseña un experimento para refutar la hipótesis. Decide cómo interpretar el resultado del experimento *a priori*, antes de realizarlo.
- Guarda un registro de todo.

La etapa de diseño del experimento es también muy sistemática. Se trata de acotar el espacio de búsqueda, ya sea reduciendo el rango de de datos a analizar o acotando la región del programa donde se encuentra el error.

Típicamente se diseña un pequeño programa que ejercita las funciones que pueden contener el error. A partir de este caso de prueba fallida se realiza una *búsqueda binaria* del error. Si el programa no funciona correctamente es porque alguno de los valores devueltos o almacenados no es el que esperábamos. Aproximadamente en el punto medio tendremos que comprobar si los valores intermedios calculados hasta el momento son los esperados. Si son correctos el error debe estar en la segunda mitad, en caso contrario estaría en la primera mitad. Repetimos este proceso hasta que encontramos el punto del error.

Es en esta etapa del proceso de depuración en la que pueden ser útiles los depuradores como *GNU debugger*. Por supuesto siempre podemos examinar los valores intermedios con un `printf` en el sitio adecuado. Pero cada vez que tengamos que examinar otro punto tendremos que recompilar el programa. El depurador, por el contrario, permite examinar y modificar cualquier dato del programa y parar la ejecución del programa en cualquier punto.

Usando GNU debugger

Para que `gdb` pueda ser aprovechado al máximo es necesario compilar el programa con soporte para depuración. Esto se especifica con la opción `-ggdb` del compilador. Típicamente se realiza añadiendo esta opción a la variable `CFLAGS` del `makefile`.

Utilizando interfaz de línea de órdenes, el primer paso consiste en invocar `gdb` especificando el ejecutable a depurar:

```
pi@raspberrypi:~/test $ gdb hola
GNU gdb (Raspbian 7.7.1+dfsg-5) 7.7.1
...
Reading symbols from hola...done.
(gdb) █
```

Muestra el *prompt* de GDB indicando que espera una orden. Muy resumidas, estas son las órdenes más frecuentes:

Orden (abreviatura)	Significado
<code>list (l)</code>	Lista el código del programa o de una función. Sin argumentos muestra desde la posición actual. El argumento puede ser una función, un archivo o incluso un rango de líneas (dos números separados por comas).
<code>break (b)</code>	Coloca un punto de ruptura en la función, línea o archivo indicado. El depurador interrumpe automáticamente la ejecución del programa al llegar a un punto de ruptura.
<code>run</code>	Ejecuta el programa con los argumentos que se indiquen hasta el siguiente punto de ruptura.
<code>print (p)</code>	Imprime el valor de variables o expresiones. Sólo se pueden visualizar aquellas expresiones que aún formen parte del contexto actual de ejecución.
<code>continue (c)</code>	Cuando se interrumpe la ejecución del programa esta orden permite continuar desde el mismo punto.
<code>next (n)</code>	Ejecuta la siguiente línea sin entrar en las funciones, es decir, las funciones se ejecutan en un sólo paso.
<code>step (s)</code>	Ejecutan la siguiente línea y si es una llamada a función accede al código de la misma.
<code>backtrace (bt)</code>	Muestra la traza de llamadas actual, es decir, las funciones que han sido invocadas y desde dónde.
<code>clear (cl)</code>	Borra punto de ruptura actual o especificado.
<code>info break (i b)</code>	Muestra los puntos de ruptura activos.
<code>quit (q)</code>	Salir de GDB.

Watchpoints

Un *watchpoint* es una indicación al depurador para interrumpir la ejecución del programa cada vez que se lea o se modifique el valor de una variable. Varias órdenes de GDB nos permiten trabajar con ellos.

Orden (abreviatura)	Significado
<code>watch var</code>	Interrumpe cuando <i>var</i> cambia.
<code>rwatch var</code>	Interrumpe cuando <i>var</i> es leída.
<code>awatch var</code>	Interrumpe cuando <i>var</i> es leída o escrita.
<code>info watch (i wat)</code>	Muestra los <i>watchpoints</i> activos.

Alterar variables y flujo

Es frecuente que en una sesión de GDB se detecte más de un error. A veces puede interesar arreglar temporalmente un problema para encontrar otro sin necesidad de recompilar. Con GDB es posible fijar valores a variables o alterar la secuencia normal de ejecución.

Orden (abreviatura)	Significado
set <i>var=valor</i>	Cambia el valor de <i>var</i> al indicado.
return <i>valor</i>	Termina la función actual y devuelve el valor indicado.

Trabajo con procesos

Algunos de nuestros programas serán multiproceso. GDB permite trabajar con múltiples procesos simultáneamente, incluso si corresponden a distintos ejecutables. Las siguientes órdenes permiten

Orden (abreviatura)	Significado
set follow-fork-mode <i>modo</i>	Si <i>modo</i> es <code>parent</code> cuando se crean nuevos procesos no se ejecutan bajo el control de GDB. Si es <code>child</code> al crearse un proceso nuevo GDB pasa a controlar al hijo en lugar de al padre.
set detach-on-fork <i>modo</i>	Si <i>modo</i> es <code>on</code> (valor por defecto) el proceso no gestionado por GDB (hijo o padre según el valor de <code>follow-fork-mode</code>) se desasocia de GDB. Si <i>modo</i> es <code>off</code> ambos procesos pasan a ser gestionados por GDB (múltiples procesos <i>inferiores</i>).
info inf (<i>i i</i>)	Muestra todos los procesos <i>inferiores</i> .
inferior <i>n</i>	Conmuta GDB al proceso inferior <i>n</i> .
attach <i>pid</i>	Asocia GDB a un proceso previamente en ejecución.
detach	Desasocia GDB del proceso en ejecución.

Trabajo con hilos

También usaremos en ocasiones los *hilos*. Se trata de un mecanismo de concurrencia mucho más eficiente que los procesos, pero sensiblemente más propenso a errores difíciles de encontrar.

Orden (abreviatura)	Significado
info threads (<i>i th</i>)	Lista de los hilos actualmente en ejecución.
thread <i>n</i> (<i>t n</i>)	Conmuta al hilo <i>n</i> .
thread apply <i>id... orden</i>	Aplica la <i>orden</i> a los hilos especificados. Si se indica <code>all</code> se aplica a todos los hilos.

Programando los periféricos

Vamos a ver ahora una serie de ejemplos de programación en C de los elementos que hemos visto en el capítulo 2. Nuestro objetivo es ofrecer un panorama de toda la gama de posibilidades que tenemos a nuestra disposición usando lenguaje C. Eso implica que no nos vamos a limitar a usar una biblioteca, sino todas las que en este momento conozco.

Cuando te enfrentas a un nuevo proyecto tienes que entender bien todos los elementos involucrados. Esto normalmente implica explorar, probar y leer. Tu primer objetivo tiene que ser reducir la incertidumbre, tener los detalles suficientes para no tener que mirar la hoja de datos de los dispositivos a cada paso cuando estemos diseñando la aplicación.

No tomes estos ejemplos como una muestra de cómo se debe programar, eso lo veremos más adelante. Se trata de que aprendas a usar los periféricos desde tu lenguaje de programación favorito. Por tanto intentamos que el código sea lo más simple y directo posible, no el más mantenible, ni siquiera el más legible. Primamos sobre todo que se vea el uso de la API ofrecida por cada biblioteca.

Entradas y salidas digitales en C

Para la programación de entradas y salidas digitales en C tenemos tres bibliotecas disponibles: [wiringPi](#) de Gordon Henderson, [bcm2835](#) de Mike McCauley y [pigpio](#) de joan@abyz.co.uk. Todas ellas son más o menos equivalentes para los propósitos del taller, aunque cada una tiene sus ventajas e inconvenientes. Para empezar te recomiendo que uses *wiringPi*. Veamos ejemplos con las tres.

Entradas y salidas digitales con *wiringPi*

Echa un vistazo al [manual de referencia de wiringPi](#) y muy especialmente a las [funciones principales](#). Explica qué hace el siguiente programa:

```
#include <wiringPi.h>
#include <unistd.h>

int main() {
    wiringPiSetupGpio();

    pinMode(18, OUTPUT);
    for(int v = 0;;v = !v) {
        digitalWrite(18, v);
        delay(1000);
    }
}
```

Para compilarlo podemos hacer un archivo `makefile` sumamente sencillo. Asumiendo que el archivo anterior se llama `test-gpio.c`:

```
LDLIBS=-lwiringPi
test-gpio: test-gpio.o
```

Compíllalo usando `make` y demuestra que funciona usando un LED en la pata J8-12, correspondiente a GPIO 18. Cuidado con no superar la corriente de 16 mA.

Si intentas ejecutarlo probablemente obtendrás un mensaje que indica que el usuario *pi* no tiene suficientes privilegios. En ese caso hay que ejecutarlo con `sudo`. Esto va a ser bastante frecuente en software que manipula dispositivos físicos.

Ningún sistema operativo serio puede permitir que un usuario normal manipule directamente los dispositivos, podría comprometer incluso la integridad física del sistema.

Un resumen de las funciones necesarias de *wiringPi* son las siguientes:

Función	Descripción
<code>wiringPiSetupGpio()</code>	Inicializa la biblioteca con la numeración de pines normal.
<code>pinMode(pin, OUTPUT)</code>	Configura un pin como salida.
<code>pinMode(pin, INPUT)</code>	Configura un pin como entrada.
<code>digitalWrite(pin, v)</code>	Saca un valor por un pin.
<code>digitalRead(pin)</code>	Lee el valor de un pin.
<code>pullUpDnControl(pin, PUD_DOWN)</code>	Configura un <i>pull-down</i> en un pin.
<code>pullUpDnControl(pin, PUD_UP)</code>	Configura un <i>pull-up</i> en un pin.
<code>delay(msec)</code>	Retardo de un número de milisegundos.

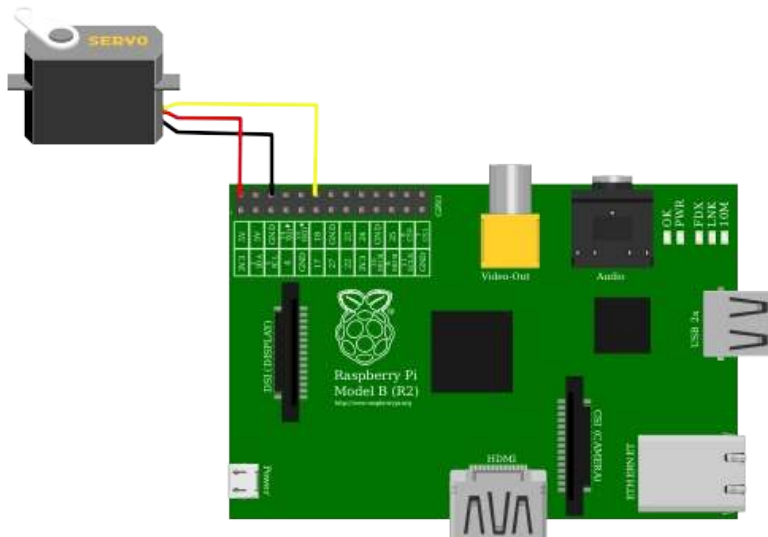
Uso de PWM

También podemos programar la generación de señales PWM con la ayuda de la biblioteca *wiringPi*. La frecuencia base para PWM en Raspberry Pi es de 19.2Mhz. Esta frecuencia puede ser dividida mediante el uso de un divisor indicado con `pwmSetClock`, hasta un máximo de 4095. A esta frecuencia funciona el algoritmo interno que genera la secuencia de pulsos, pero en el caso del BCM2835 se dispone de dos modos de funcionamiento, un modo equilibrado (*balanced*) en el que es difícil controlar la anchura de los pulsos, pero permite un control PWM de muy alta frecuencia, y un modo *mark and space* que es mucho más intuitivo y más apropiado para controlar servos. El modo *balanced* es apropiado para controlar la potencia suministrada a la carga o para transmisión de información.

En el modo *mark and space* el módulo PWM incrementará un contador interno hasta llegar a un límite configurable, el rango de PWM, que puede ser de como máximo 1024. Al comienzo del ciclo el pin se pondrá a 1 lógico, y se mantendrá hasta que el contador interno llegue al valor puesto por el usuario. En ese momento el pin se pondrá a 0 lógico y se mantendrá hasta el final del ciclo.

Veamos su aplicación al control de un servomotor. Un servomotor tiene una entrada de señal para indicar la inclinación deseada. Cada 20ms espera un pulso y la anchura de este pulso determina la inclinación del servo. Alrededor de 1.5ms es la anchura del pulso necesaria para la posición centrada. Una anchura menor hace girar el servo en sentido antihorario (hasta 1ms aproximadamente) y una duración mayor lo hace girar en sentido horario (hasta 2ms aproximadamente). En este caso hay que calcular el rango y el divisor para que el pulso se produzca cada 20ms y el control de la anchura del pulso alrededor de los 1.5ms sea con la máxima resolución posible.

El montaje es tal como muestra la figura. El cable rojo del servo (V+) se conecta a +5V en P1-2 o P1-4, el cable negro o marrón (V-) a GND en P1-6 y el cable amarillo, naranja o blanco (signal) a GPIO18 en P1-12. No se necesita ningún otro componente.



```
#include <wiringPi.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if (argc < 5) {
        printf("Usage: %s divisor rango min max\n", argv[0]);
        exit(0);
    }

    int div = atoi(argv[1]);
    int range = atoi(argv[2]);
    int min = atoi(argv[3]);
    int max = atoi(argv[4]);

    wiringPiSetupGpio();

    pinMode(18, PWM_OUTPUT);
    pwmSetMode(PWM_MODE_MS);
    pwmSetClock(div);
    pwmSetRange(range);

    for(;;) {
        pwmWrite(18, min);
        delay(1000);
        pwmWrite(18, max);
        delay(1000);
    }
}
```

Para tener máximo control de la posición del servo probaremos con el rango máximo, de 1024. En ese caso el divisor tiene que ser tal que la frecuencia del pulso PWM sea:

$$f = \frac{f_{base}}{rango \times div} = \frac{19.2 \times 10^6 Hz}{1024 \times div} = \frac{1}{20ms} = 50Hz$$

Es decir, el divisor habría que configurarlo a 390. El rango completo del servo depende del modelo concreto. Teóricamente debería ser entre 52 y 102, siendo el valor completamente centrado 77. En la práctica habrá que probar el servo concreto, nuestros experimentos dan un rango útil entre 29 y 123 para el microservo de TowerPro disponible en el laboratorio.

Función	Descripción
<code>pinMode(pin, PWM_OUTPUT)</code>	Configura un pin como salida PWM.
<code>pwmSetMode(PWM_MODE_MS)</code>	Configura el modo PWM a <i>mark-and-space</i> .
<code>pwmSetClock(div)</code>	Configura el divisor del reloj.
<code>pwmSetRange(rango)</code>	Configura el rango del pulso.
<code>pwmWrite(pin, v)</code>	Configura la anchura del pulso.

Programación de entradas y salidas digitales con *bcm2835*

La biblioteca *bcm2835* es un envoltorio muy fino de las capacidades del hardware. Es decir, prácticamente describe en C lo que aparece en la hoja de datos de Broadcom. En este sentido es ideal para explorar la arquitectura y extraer el máximo jugo de tu Raspberry Pi. Para tareas no triviales no tendrás más remedio que estudiar bien [Citation not found] para entender el funcionamiento.

En cuanto al manejo de entradas y salidas digitales la biblioteca *bcm2835* tiene una interfaz muy similar a *wiringPi* pero soporta muchas más capacidades del hardware subyacente. En este taller no usaremos las características avanzadas.

Una característica interesante de los programas que usan *bcm2835* es que no requieren ser ejecutados como superusuario si solo acceden a los pines de GPIO, basta con que el usuario pertenezca al grupo `gpio`.

```
#include <bcm2835.h>

int main() {
    bcm2835_init();
    bcm2835_gpio_fsel(18, BCM2835_GPIO_FSEL_OUTP);
    for(int v = 0;;v = !v) {
        bcm2835_gpio_write(18, v);
        bcm2835_delay(1000);
    }
    bcm2835_close();
}
```

Como puede apreciarse el código es prácticamente equivalente a *wiringPi*. La compilación es también similar. Para compilarlo podemos hacer un archivo `makefile` casi equivalente al ejemplo de *wiringPi*.

```
CFLAGS=-I/usr/local/include
LDFLAGS=-L/usr/local/lib
LDLIBS=-lbcm2835
test-gpio: test-gpio.o
```

Al margen del nombre de la biblioteca vemos que es preciso indicar que busque archivos de cabecera en `/usr/local/include` y busque las bibliotecas en `/usr/local/lib`. Esto se debe a que *bcm2835* todavía no está como un paquete del sistema, y la hemos instalado de forma manual.

Veamos un resumen de las funciones más importantes.

Función	Descripción
<code>bcm2835_init()</code>	Inicializa la biblioteca.
<code>bcm2835_close()</code>	Libera los recursos empleados por la biblioteca.
<code>bcm2835_gpio_fsel(pin, BCM2835_GPIO_FSEL_OUTP)</code>	Configura un pin como salida.
<code>bcm2835_gpio_fsel(pin, BCM2835_GPIO_FSEL_INPT)</code>	Configura un pin como entrada.
<code>bcm2835_gpio_write(pin, v)</code>	Saca un valor por un pin.
<code>bcm2835_gpio_lev(pin)</code>	Lee el valor de un pin.
<code>bcm2835_gpio_set_pud(pin, BCM2835_GPIO_PUD_DOWN)</code>	Configura un <i>pull-down</i> en un pin.
<code>bcm2835_gpio_set_pud(pin, BCM2835_GPIO_PUD_UP)</code>	Configura un <i>pull-up</i> en un pin.
<code>bcm2835_delay(msec)</code>	Retardo de un número de milisegundos.
<code>bcm2835_delayMicroseconds(usec)</code>	Retardo de un número de microsegundos.

Entre las características avanzadas *bcm2835* implementa la posibilidad de cambiar el valor de un conjunto de pines de golpe, de leer un conjunto de pines de golpe, mejor control sobre los eventos de flanco de subida, bajada o cambio de nivel, etc.

PWM con *bcm2835*

Al programar la modulación de anchura de pulsos *bcm2835* requiere conocer un poco del funcionamiento del hardware. Por ejemplo, requiere saber que los dos canales PWM no están asociados a un único pin y que se puede asociar uno u otro canal a algunos de los pines seleccionando funciones alternativas concretas. Siguiendo el mismo ejemplo de *wiringPi* configuraremos el pin GPIO18 como salida PWM. Para ello tendremos que seleccionar la función alternativa 5 de ese pin, que corresponde al canal PWM0. ¿Entiendes ahora la utilidad del *flyer* que te damos en el taller? A partir de ese momento solo trabajamos con el canal, no con el pin.

```
#include <bcm2835.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc < 5) {
        printf("Usage: %s divisor rango min max\n", argv[0]);
        exit(0);
    }

    int div = atoi(argv[1]);
    int range = atoi(argv[2]);
    int min = atoi(argv[3]);
    int max = atoi(argv[4]);

    bcm2835_init();
    bcm2835_gpio_fsel(18, BCM2835_GPIO_FSEL_ALT5);
    bcm2835_pwm_set_clock(div);
    bcm2835_pwm_set_mode(0, 1, 1);
    bcm2835_pwm_set_range(0, range);

    for(;;) {
        bcm2835_pwm_set_data(0, min);
        bcm2835_delay(1000);
        bcm2835_pwm_set_data(0, max);
        bcm2835_delay(1000);
    }
    bcm2835_close();
    return 0;
}
```

Resumiendo, las siguientes funciones nos permiten trabajar con la modulación PWM:

Función	Descripción
<code>bcm2835_gpio_fsel(pin, BCM2835_GPIO_FSEL_ALTn)</code>	Selecciona la función alternativa <i>n</i> del pin.
<code>bcm2835_pwm_set_clock(div)</code>	Configura el divisor para el reloj de 19.2MHz.
<code>bcm2835_pwm_set_mode(canal, ms, activo)</code>	Configura el modo del canal PWM y/o lo activa.
<code>bcm2835_pwm_set_range(canal, range)</code>	Configura el rango del pulso en un canal PWM.
<code>bcm2835_pwm_set_data(canal, v)</code>	Configura la anchura de pulso del canal PWM.

Programación de entradas y salidas digitales con *pigpio*

La biblioteca *pigpio* está a caballo entre las dos anteriores. Por un lado implementa una interfaz de bajo nivel que prácticamente reproduce la hoja de datos de Broadcom en C. Por otro lado implementa capas de abstracción que amplían la funcionalidad considerablemente. Así por ejemplo añade la posibilidad de simular modulación PWM en cualquier pin e incorpora capacidades de depuración muy interesantes.

Desde el punto de vista de la programación de entradas y salidas digitales es muy similar a las otras bibliotecas.

```
#include <pigpio.h>

int main() {
    gpioInitialise();
    gpioSetMode(18, PI_OUTPUT);
    for(int v = 0; v != 1; v++) {
        gpioWrite(18, v);
        gpioDelay(1000000);
    }
    gpioTerminate();
}
```

Prácticamente idéntico a los demás ejemplos salvo por los nombres de las funciones. La compilación es muy similar a la de los ejemplos de *bcm2835* porque la biblioteca *pigpio* tampoco está disponible como paquete del sistema y hemos tenido que instalarla de forma manual.

```
CFLAGS=-I/usr/local/include -pthread
LDFLAGS=-L/usr/local/lib -pthread
LDLIBS=-lpigpio -pthread
test-gpio: test-gpio.o
```

Una diferencia importante con respecto a las demás bibliotecas es que hay que habilitar el uso de hilos y añadir la biblioteca de hilos del sistema. Esto es necesario porque muchas de las capacidades añadidas de *pigpio* se implementan como hilos.

Función	Descripción
<code>gpioInitialise()</code>	Inicializa la biblioteca.
<code>gpioTerminate()</code>	Libera los recursos empleados por la biblioteca.
<code>gpioSetMode(pin, PI_OUTPUT)</code>	Configura un pin como salida.
<code>gpioSetMode(pin, PI_INPUT)</code>	Configura un pin como entrada.
<code>gpioWrite(pin, v)</code>	Saca un valor por un pin.
<code>gpioRead(pin)</code>	Lee el valor de un pin.
<code>gpioSetPullUpDown(pin, PI_PUD_DOWN)</code>	Configura un <i>pull-down</i> en un pin.
<code>gpioSetPullUpDown(pin, PI_PUD_UP)</code>	Configura un <i>pull-up</i> en un pin.
<code>gpioDelay(usec)</code>	Retardo de un número de microsegundos.

Desde el punto de vista de diseño la interfaz *pigpio* está mejor diseñada que *wiringPi* pero en los ejemplos del taller no lo vas a notar. Lo notarás por ejemplo cuando uses `gpioSetAlertFunc` de *pigpio* en lugar de `wiringPiISR` de *wiringPi*.

Modulación PWM con *pigpio*

La modulación PWM con *pigpio* puede hacerse con funciones de bajo nivel de forma equivalente a como se hace en *bcm2835* pero soporta además una interfaz de alto nivel mucho más sencilla. Existe una función para controlar servos con PWM y otra para controlar el *duty-cycle* y con ello la potencia entregada a una carga.

Veamos el ejemplo usando esta interfaz:

```
#include <pigpio.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc < 5) {
        printf("Usage: %s min max\n", argv[0]);
        exit(0);
    }

    int min = atoi(argv[1]);
    int max = atoi(argv[2]);

    gpioInitialise();
    for(;;) {
        gpioServo(18, min);
        gpioDelay(1000000);
        gpioServo(18, max);
        gpioDelay(1000000);
    }
    gpioTerminate();
    return 0;
}
```

No es necesario especificar los parámetros de bajo nivel, tan solo la anchura del pulso en milisegundos. Un ancho de cero para la señal PWM. El resto de valores válidos están entre 500 y 2500, aunque el rango real del servo depende del modelo concreto.

Otra característica interesante de pigpio es que estas funciones se pueden aplicar sobre cualquier pin. Si es uno de los que soportan PWM por hardware lo utilizará de forma transparente y si no lo simulará con un hilo independiente. La frecuencia de la señal PWM que genera es en todos los casos de 50Hz.

Un resumen de las funciones de alto nivel en pigpio:

Función	Descripción
<code>gpioServo(pin, ancho)</code>	Señal PWM para controlar servo con una anchura de pulso dada.
<code>gpioPWM(pin, duty)</code>	Genera una señal PWM con un <i>duty-cycle</i> determinado (hasta <i>rango</i>).
<code>gpioSetPWMrange(pin, rango)</code>	Cambia el rango de la señal PWM generada.
<code>gpioSetPWmfrequency(pin, f)</code>	Cambia la frecuencia de la señal PWM generada.

Ejercicios

No intentes usar todas las bibliotecas de golpe. Elige una y espera a sentirte cómodo con ella para probar otra. De momento te proponemos los siguientes ejercicios.

1. Configura y programa el hardware y el software necesario para tener dos LEDs parpadeando al mismo ritmo pero manteniendo solo uno de ellos encendido a la vez.
2. Modifica el ejemplo anterior para que la conmutación se produzca solamente cuando se aprieta un pulsador. Uno de los LEDs estará encendido cuando el pulsador no esté apretado, el otro estará encendido solamente cuando el pulsador esté apretado.

Retos para la semana

1. **Moderado** Diseña un mecanismo para poder controlar una matriz de LEDs de como mínimo 16x32 con la Raspberry Pi.

Programación de la interfaz I2C

La programación precisa de un dispositivo I2C depende mucho del fabricante. En general necesitaremos primero una configuración y luego entraremos en un bucle que lee o escribe datos. Por ejemplo, veamos cómo se maneja el acelerómetro MPU6050.

Programación I2C con *wiringPi*

Como siempre *wiringPi* sacrifica flexibilidad en aras de una mayor simplicidad. Ni siquiera proporciona una función para cambiar la frecuencia del reloj y no soporta transferencias de bloques, sino solo de registros de 8 y 16 bits.

```
#include <wiringPiI2C.h>
#include <stdio.h>
#include <stdlib.h>

short swap(short n) { return (n << 8) | (n >> 8) & 0xff; }

int main()
{
    int fd = wiringPiI2CSetup(0x68);
    if (0x68 != wiringPiI2CReadReg8(fd, 117)) {
        puts("No se encuentra el dispositivo");
        return 1;
    }
    wiringPiI2CWriteReg8(fd, 107, 0);
    for(;;) {
        short t = wiringPiI2CReadReg16(fd, 65);
        printf("temperatura: %f\n", swap(t)/340. + 36.53);

        short ax = wiringPiI2CReadReg16(fd, 59);
        short ay = wiringPiI2CReadReg16(fd, 61);
        short az = wiringPiI2CReadReg16(fd, 63);
        printf("ax=%d, ay=%d, az=%d\n", ax, ay, az);
        delay(500);
    }
    close(fd);
    return 0;
}
```

La función de inicialización `wiringPiI2CSetup` necesita la dirección del dispositivo, que podemos obtener con `i2cdetect` y devuelve un descriptor de archivo que hay que usar en todas las demás llamadas relacionadas con ese dispositivo. Podemos hacer transferencias de lectura y escritura de/a registros de 8 y 16 bits. Típicamente se configura previamente el dispositivo con llamadas a `wiringPiI2CWriteReg8` según la hoja de datos del fabricante.

Un resumen de las funciones empleadas sería:

Función	Descripción
<code>wiringPiI2CSetup(dir)</code>	Inicializa un canal I2C.
<code>wiringPiI2CWriteReg8(fd, reg, v)</code>	Escribe un registro de 8 bits.
<code>wiringPiI2CWriteReg16(fd, reg, v)</code>	Escribe un registro de 16 bits.
<code>wiringPiI2CReadReg8(fd, reg)</code>	Lee un registro de 8 bits.
<code>wiringPiI2CReadReg16(fd, reg)</code>	Lee un registro de 16 bits.
<code>close(fd)</code>	Cierra el canal.

Programación de I2C con *bcm2835*

En *bcm2835* la interfaz es también sencilla pero no se requiere un descriptor de archivo para cada dispositivo I2C y la comunicación es un poco engorrosa, porque el direccionamiento de los registros debe hacerse de forma manual con un `write`.

```
#include <bcm2835.h>
#include <stdio.h>

short swap(short n) { return (n << 8) | (n >> 8) & 0xff; }

int main()
{
    bcm2835_init();
    bcm2835_i2c_begin();
    bcm2835_i2c_setSlaveAddress(0x68);
    bcm2835_i2c_set_baudrate(100000);
    char dir = 117;
    bcm2835_i2c_write(&dir, 1);
    bcm2835_i2c_read(&dir, 1);
    if (0x68 != dir) {
        puts("No se encuentra el dispositivo");
        return 1;
    }
    char cfg[] = {107, 0};
    bcm2835_i2c_write(cfg, 2);
    for(;;) {
        dir = 59;
        bcm2835_i2c_write(&dir, 1);
        short data[7];
        bcm2835_i2c_read(data, 14);
        printf("temperatura: %f\n", swap(data[3])/340. + 36.53);
        printf("ax=%d, ay=%d, az=%d\n",
            swap(data[0]), swap(data[1]), swap(data[2]));
        delay(500);
    }
    bcm2835_i2c_end();
    bcm2835_close();
    return 0;
}
```

Un resumen de las funciones utilizadas.

Función	Descripción
<code>bcm2835_i2c_begin()</code>	Inicializa las comunicaciones I2C.
<code>bcm2835_i2c_end()</code>	Libera los recursos ocupados por <i>begin</i> .
<code>bcm2835_i2c_setSlaveAddress(addr)</code>	Asigna la dirección destino.
<code>bcm2835_i2c_set_baudrate(rate)</code>	Configura la velocidad del reloj I2C.
<code>bcm2835_i2c_write(buf, len)</code>	Escribe un conjunto de bytes en ráfaga.
<code>bcm2835_i2c_read(buf, len)</code>	Lee un conjunto de bytes en ráfaga.

En realidad la biblioteca soporta alguna otra función para tratar con casos especiales. Si te enfrentas a un módulo I2C nuevo consulta la documentación de *bcm2835* por si te facilita las cosas.

Programación de I2C con *pigpio*

La biblioteca *pigpio* implementa una interfaz que combina las dos aproximaciones anteriores. Puede leer o escribir bytes o palabras, pero también bloques, lo que simplifica sensiblemente el programa. Además permite acceder a los dos buses I2C.

```

#include <pigpio.h>
#include <stdio.h>

short swap(short n) { return (n << 8) | (n >> 8) & 0xff; }

int main()
{
    gpioInitialise();
    int i2c = i2cOpen(1, 0x68, 0);
    if (0x68 != i2cReadByteData(i2c, 117)) {
        puts("No se encuentra el dispositivo");
        return 1;
    }
    i2cWriteByteData(i2c, 107, 0);
    for(;;) {
        short data[7];
        i2cReadI2CBlockData(i2c, 59, (char*)data, 14);
        printf("temperatura: %f\n", swap(data[3])/340. + 36.53);
        printf("ax=%d, ay=%d, az=%d\n",
            swap(data[0]), swap(data[1]), swap(data[2]));
        delay(500);
    }
    i2cClose(i2c);
    gpioTerminate();
    return 0;
}

```

Soporta muchas más funciones, pero habitualmente no necesitaremos más de estas.

Función	Descripción
<code>i2cOpen(bus, dir, 0)</code>	Abre un canal I2C para el dispositivo <i>dir</i> .
<code>i2cClose(h)</code>	Cierra el canal asociado a <i>h</i> .
<code>i2cWriteByteData(h, reg, v)</code>	Escribe un byte <i>v</i> en un registro <i>reg</i> .
<code>i2cWriteWordData(h, reg, v)</code>	Escribe un short <i>v</i> en un registro <i>reg</i> .
<code>i2cReadByteData(h, reg)</code>	Lee un byte de un registro <i>reg</i> .
<code>i2cReadWordData(h, reg)</code>	Lee un short de un registro <i>reg</i> .
<code>i2cWriteI2CBlockData(h, reg, buf, len)</code>	Escribe un bloque a partir de <i>reg</i> .
<code>i2cReadI2CBlockData(h, reg, buf, len)</code>	Lee un bloque a partir de <i>reg</i> .

Como en el resto de módulos *pigpio* es la más completa de las tres, y *bcm2835* la más cercana al hardware y por tanto la más pequeña. Nuestra recomendación personal es que utilices *wiringPi* para empezar, por su mayor simplicidad.

Programación en C de SPI

Las tres bibliotecas que hemos usado para la programación de entradas y salidas digitales soportan también el uso de la interfaz SPI. Los módulos SPI se benefician de la capacidad de envío y recepción concurrente y pueden conseguirse tasas muy razonables (30MHz).

Programación de SPI con *wiringPi*

La programación es sencilla en cuanto que solo utiliza dos funciones, pero puede ser realmente enrevesada de entender la comunicación con algunos dispositivos SPI. El motivo es que en SPI para poder leer datos hay que escribir datos, de hecho se lee a la vez que se escribe. Esto hace que en los dispositivos reales haya que hacer muchas transacciones que se descartan por completo.

Este ejemplo reproduce el que describíamos en el capítulo anterior utilizando la herramienta `pigs`, esta vez empleando *wiringPi*.

```
#include <stdio.h>
#include <stdlib.h>
#include <wiringPiSPI.h>

void ads1118_rw(int ch, char buf[4]) {
    buf[0] = 0x80; buf[1] = 0x4b;
    buf[0] = 0x80; buf[1] = 0x4b;
    wiringPiSPIDataRW (0, buf, 4);
}

int main (void) {
    wiringPiSPISetup(0, 4000000);

    char data[4];
    ads1118_rw(0, data);
    printf("Control reg = %02x%02x\n", data[2], data[3]);
    delay(100);
    ads1118_rw(0, data);
    short v = data[0] << 8 | data[1];
    printf("AD0 = %02x%02x (%d)\n", data[0], data[1], v);
    return 0;
}
```

La función `wiringPiSPISetup` inicializa la comunicación para el canal 0 a 10Mz. Hay dos canales disponibles (0 y 1) que utilizan las mismas patas salvo la de selección `SPI_CE0` y `SPI_CE1` respectivamente.

Las llamadas a `wiringPiSPIDataRW` realizan una transacción SPI donde se escribe y se lee de manera concurrente un conjunto de bytes. El significado preciso de lo que se lee y se escribe depende del dispositivo y en algunos casos puede requerir descartar parte o toda la información. En este caso el primer byte es la orden y a continuación se envían los argumentos.

Un resumen de las funciones involucradas:

Función	Descripción
<code>wiringPiSPISetup(canal, velocidad)</code>	Prepara un canal SPI y lo configura a una velocidad determinada.
<code>wiringPiSPIDataRW(canal, bytes, long)</code>	Escribe y lee a la vez un conjunto de bytes.

La simplicidad es máxima pero también se pierden las capacidades del hardware para poder acomodar todos los modos de transferencia de SPI. Pueden consultarse más detalles en el artículo de Gordon Henderson disponible en projects.drogon.net.

El problema de *wiringPi* es que para transferencias SPI no permite seleccionar el modo. Su autor considera que la gran mayoría de los módulos SPI usan el modo 0. Puede ser cierto, pero los ADS1118 que usamos en el taller son modo 1.

Warning La biblioteca *wiringPi* asume que utilizamos SPI0 con el modo 0. Por este motivo no la recomendamos, porque el módulo CJMCU-1118 utiliza el modo 1.

Programación de SPI con *bcm2835*

En la biblioteca *bcm2835* contamos con una gama de funciones más próxima al hardware. Algunas funciones son similares a *wiringPi* pero añade muchas más para configurar el modo de transferencia y la interfaz SPI.

```
#include <stdio.h>
#include <bcm2835.h>

int main (void) {
    bcm2835_init();
    bcm2835_spi_begin();
    bcm2835_spi_setDataMode(1);
    bcm2835_spi_setClockDivider(5); // 19.2MHz / 5 ~ 4MHz
    bcm2835_spi_chipSelect(0);

    char tdata[4] = { 0x80, 0x4b, 0x80, 0x4b };
    char rdata[4];
    bcm2835_spi_transfernb(tdata, rdata, 4)
    printf("Control reg = %02x%02x\n", rdata[2], rdata[3]);
    bcm2835_delay(100);
    bcm2835_spi_transfernb(tdata, rdata, 4)
    short v = rdata[0] << 8 | rdata[1];
    printf("AD0 = %02x%02x (%d)\n", rdata[0], rdata[1], v);

    bcm2835_spi_end();
    bcm2835_close();
    return 0;
}
```

Aunque parece más complejo realmente se debe al mayor control de la inicialización. La inicialización es más larga, pero la posibilidad de usar buffers distintos de transmisión y recepción simplifica muchos casos frecuentes.

Un resumen de las funciones involucradas:

Función	Descripción
<code>bcm2835_spi_begin()</code>	Inicializa el módulo de SPI.
<code>bcm2835_spi_end()</code>	Libera los recursos empleados en la inicialización.
<code>bcm2835_spi_setClockDivider(div)</code>	Define el divisor del reloj.
<code>bcm2835_spi_setDataMode(modo)</code>	Modo SPI según se comentó en capítulo 2.
<code>bcm2835_spi_chipSelect(cs)</code>	Pin de CS (0, 1, 2), 3 = ninguno.
<code>bcm2835_spi_setChipSelectPolarity(cs, v)</code>	Define la polaridad del pin <i>cs</i> como activa a valor <i>v</i> .
<code>bcm2835_spi_transfer(v)</code>	Transmite y recibe (devuelve) un byte.
<code>bcm2835_spi_transfernb(tbuf, rbuf, len)</code>	Transmite y recibe <i>len</i> bytes.
<code>bcm2835_spi_transfern (buf, len)</code>	Transmite y recibe <i>len</i> bytes en el mismo buffer.
<code>bcm2835_spi_writenb (buf, len)</code>	Transmite <i>len</i> bytes.

Programación de SPI con *pigpio*

Con *pigpio* es posible utilizar el periférico auxiliar SPI (SPI1) además del principal (ver bandera *A* del campo de banderas en *spiOpen*). La ventaja es que esta otra interfaz SPI tiene tamaño de palabra configurable y tres líneas de *chip select* disponibles (en lugar de dos). Por otro lado la interfaz principal es sensiblemente más rápida, por lo que utilizaremos esa normalmente.

```
#include <stdio.h>
#include <pigpio.h>

int main (void) {
    gpioInitialise();
    int spi = spiOpen(0, 4000000, 1);
    char tdata[4] = { 0x80, 0x4b, 0x80, 0x4b };
    char rdata[4];
    spiXfer(spi, tdata, rdata, 4)
    printf("Control reg = %02x%02x\n", rdata[2], rdata[3]);
    bcm2835_delay(100);
    spiXfer(spi, tdata, rdata, 4)
    short v = rdata[0] << 8 | rdata[1];
    printf("AD0 = %02x%02x (%d)\n", rdata[0], rdata[1], v);
    spiClose(spi);
    gpioTerminate();
    return 0;
}
```

Al igual que *bcm2835* utiliza buffers diferentes para la transmisión y la recepción, lo que en muchos casos simplifica el trabajo y permite usar buffers constantes en la transmisión.

Función	Descripción
<code>spiOpen(ch, b, f)</code>	Abre un canal SPI con una frecuencia <i>b</i> y banderas <i>f</i> (ver capítulo 2).
<code>spiClose(spi)</code>	Cierra canal SPI.
<code>spiRead(spi, buf, n)</code>	Lee <i>n</i> bytes del canal SPI.
<code>spiWrite(spi, buf, n)</code>	Escribe <i>n</i> bytes por el canal SPI.
<code>spiXfer(spi, tbuf, rbuf, n)</code>	Lee y escribe simultáneamente <i>n</i> bytes del canal SPI.

Medir tiempos de forma precisa

La medición del tiempo de descarga de un condensador se ha propuesto como técnica para [medir magnitudes analógicas usando las entradas digitales](#) de la Raspberry Pi. La propuesta de Adafruit utiliza el número de iteraciones de un bucle para medir el tiempo. Como ellos mismos reconocen esto no es muy preciso.

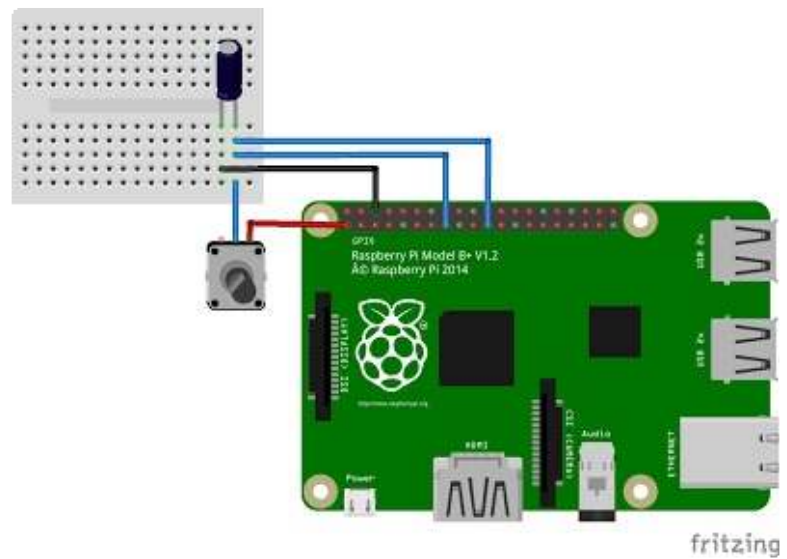
Raspbian es un sistema de tiempo compartido. La Raspberry Pi ejecuta varios programas a la vez y esto implica que el procesador puede [desalojar](#) nuestro programa para ejecutar otro programa. En ese caso el número de iteraciones del bucle será sensiblemente menor de lo normal. Pero, ¿cómo sabemos si ha habido desalojo? La triste realidad es que un programa de usuario no puede saberlo, no tiene control sobre esto. Ni siquiera es ésta la única causa de incertidumbre, puede haber interferencia de los manejadores de interrupción, de los manejadores de dispositivo, etc.

Pero hay una forma de medir el tiempo con bastante precisión, usando la pata MISO de la interfaz SPI y una pata de salida digital.

Supongamos que tenemos un dispositivo sensor cuya medida se materializa en el valor de una resistencia. Puede tratarse de una LDR, como en el caso del artículo de Adafruit, o de un simple potenciómetro, o un termistor, o un sensor piezoresistivo, o un sensor magnetorresistivo, ... Construimos un circuito RC similar a la figura.

Descargamos el condensador poniendo la pata GPIO22 a nivel bajo durante un tiempo suficiente. Configuramos la pata GPIO22 como entrada para que quede en alta impedancia y empezamos una transferencia SPI de gran tamaño. El buffer debe estar lleno de ceros hasta que la carga del condensador es suficiente para poder ser interpretada como un 1. El primer byte distinto de cero marca el instante de tiempo en que el condensador está razonablemente cargado. Este tiempo es proporcional a RC y por tanto a R. Se puede realizar un calibrado para medir con precisión absoluta, pero en cualquier caso tenemos una medida precisa que nos permite comparar.

El código es sumamente sencillo. Lo proporcionamos solamente en *wiringPi*.



Montaje para la medición precisa de una resistencia.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wiringPi.h>
#include <wiringPiSPI.h>

enum {
    BUF_SIZE = 4096,
    DISCHARGE = 22
};

int main (void) {
    wiringPiSPISetup(0, 500000);

    char* buf = malloc(BUF_SIZE);

    wiringPiSetupGpio();
    pinMode(DISCHARGE, OUTPUT);
    digitalWrite(DISCHARGE, 0);
    sleep(1);
    pinMode(DISCHARGE, INPUT);

    wiringPiSPIDataRW (0, buf, BUF_SIZE);
    for (int i = 0; i < BUF_SIZE; ++i)
        if (buf[i] != 0) {
            printf("%d %02x\n", i, buf[i]);
            break;
        }
    free(buf);
    return 0;
}
```


El programa imprime la posición del buffer donde la entrada empieza a ser distinta de cero y el primer valor distinto de cero. Esa posición habría que multiplicarla por 8 para traducirla a ciclos de SCLK y podría ajustarse con el valor para obtener el número exacto de ciclos.

La precisión depende del periodo de reloj empleado. En el ejemplo hemos empleado un reloj de 500KHz, pero puede subirse hasta 32MHz con seguridad. El problema es que a mayor reloj, mayor es el buffer que tenemos que usar en la transferencia SPI.

Para buffers mayores de 4KB hay que especificarlo en la línea de órdenes del kernel (`/boot/cmdline.txt`) y reiniciar la Raspberry Pi. Por ejemplo, para 256KB se añadiría:

```
spidev.bufsiz=262144
```

Puede mejorarse el código haciendo búsqueda por bisección. Es deliberadamente simple para que se entienda desde el punto de vista conceptual. El resultado es que podríamos medir RC con una precisión de hasta 1/32 us. Si usamos un condensador de 1uF esto implica que podemos medir R con una precisión de 1/32 Ohm. Incluso si usamos el reloj de 500KHz tendremos una precisión de 2 Ohm, que tampoco está nada mal.

Los resultados reales pueden ser algo peores por *jitter* o inestabilidad en SCLK o ruido en la resistencia. Un condensador cerámico en paralelo con el electrolítico puede ayudar a quitar ruido de alta frecuencia. En cualquier caso el método es mucho más preciso que la propuesta original de Adafruit, y no depende del estado de carga del sistema.

Retos para la semana

1. **Fácil** Diseña un mecanismo para poder controlar tiras de LEDs empleando la interfaz SPI.

Comunicaciones en red

Tanto los modelos B+ como la 2B y la 3B incluyen interfaz Ethernet. La Raspberry Pi 3 modelo B que utilizamos en este taller incluye WiFi pero cualquiera de las otras puede tener también WiFi por un precio de unos 4€ empleando una interfaz WiFi USB. Por tanto cualquier proyecto de Raspberry Pi debe plantearse la posibilidad de comunicar datos a través de una red TCP/IP.

Para programar en red en GNU/Linux, como en la mayoría de los sistemas operativos modernos, se utiliza una interfaz de programación denominada *socket API*. Se trata de un conjunto de funciones diseñadas para que la programación de redes se parezca mucho a la entrada/salida con archivos.

En el capítulo 2 ya hemos introducido la interfaz *socket*. En C se incluye esta interfaz en la biblioteca del sistema `libc` que se incorpora automáticamente. La forma de usar esta biblioteca es prácticamente lo que hemos visto en el capítulo 2.

Comunicaciones UDP

El siguiente ejemplo muestra el servidor UDP más simple posible.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <netdb.h>
#include <arpa/inet.h>

static int udp_server_socket(const char* service);

int main() {
    int fd = udp_server_socket("9999");
    for(;;) {
        char buf[1024];
        int n = read(fd, buf, sizeof(buf));
        assert(n >= 0);
        if (n == 0) break;
        buf[n]='\0';
        printf("%s", buf);
    }
    close(fd);
    return 0;
}
```

La función `udp_server_socket` simplemente crea un nuevo *socket* UDP con los parámetros indicados y llama a *bind* para fijar el puerto donde escucha mensajes.

```
static struct sockaddr_in ip_address(const char* host,
                                     const char* service,
                                     const char* proto);

static int udp_server_socket(const char* service)
{
    struct sockaddr_in sin = ip_address("0.0.0.0", service, "udp");
    struct protoent* pe = getprotobyname("udp");
    assert(pe != NULL);
    int fd = socket(PF_INET, SOCK_DGRAM, pe->p_proto);
    assert (fd >= 0);
    assert (bind(fd, (struct sockaddr*)&sin, sizeof(sin)) >= 0);
    return fd;
}
```

La función `ip_address` construye una estructura que representa una dirección completa de un servicio IP (host, puerto y protocolo).

```
static struct sockaddr_in ip_address(const char* host,
                                    const char* service,
                                    const char* proto)
{
    struct sockaddr_in sin;
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    struct hostent* he = gethostbyname(host);
    if (he != NULL)
        memcpy(&sin.sin_addr, he->h_addr_list[0], he->h_length);
    else
        sin.sin_addr.s_addr = inet_addr(host);
    struct servent* se = getservbyname(service, proto);
    sin.sin_port = (se != NULL? se->s_port : htons(atoi(service)));
    return sin;
}
```

Mete estas tres funciones en un archivo y compílalo. Prueba que funciona usando un cliente *netcat*.

El cliente correspondiente en C es similar.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <netdb.h>
#include <arpa/inet.h>

static int udp_client_socket(const char* host, const char* service);

int main() {
    int fd = udp_client_socket("localhost", "9999");
    for(;;) {
        char buf[1024];
        fgets(buf, sizeof(buf), stdin);
        int n = write(fd, buf, strlen(buf));
        assert(n >= 0);
        if (n == 0) break;
    }
    close(fd);
    return 0;
}
```

La función `udp_client_socket` es similar a `udp_server_socket` pero en lugar de llamar a *bind* llama a *connect* para fijar la dirección destino de los mensajes.

```
static struct sockaddr_in ip_address(const char* host,
                                    const char* service,
                                    const char* proto);

static int udp_client_socket(const char* host, const char* service)
{
    struct sockaddr_in sin = ip_address(host, service, "udp");
    struct protoent* pe = getprotobyname("udp");
    assert(pe != NULL);
    int fd = socket(PF_INET, SOCK_DGRAM, pe->p_proto);
    assert (fd >= 0);
    assert (connect(fd, (struct sockaddr*)&sin, sizeof(sin)) >= 0);
    return fd;
}
```

La función `ip_address` ya la comentamos en el servidor. Construye el cliente y prueba que funciona correctamente con *netcat*. Después haz la prueba de integración con servidor y cliente.

Comunicaciones TCP

Desde el punto de vista de la programación la diferencia fundamental con el caso anterior radica en la necesidad de establecer conexiones para realizar una comunicación. Esto complica algo el servidor.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <netdb.h>
#include <arpa/inet.h>

static int tcp_master_socket(const char* service);

int main() {
    int master = tcp_master_socket("9999");
    for(;;) {
        int fd = accept(master, (struct sockaddr*) NULL, NULL);
        assert (fd >= 0);
        for(;;) {
            char buf[1024];
            int n = read(fd, buf, sizeof(buf));
            assert(n >= 0);
            if (n == 0) break;
            buf[n]='\0';
            printf("%s", buf);
        }
        close(fd);
    }
    close(master);
    return 0;
}

```

Ahora el socket maestro simplemente se usa para crear nuevos sockets esclavos cuando ocurre una conexión con *accept*. El esclavo se utiliza para manejar los datos relativos a esa conexión.

La función `tcp_master_socket` simplemente crea el socket TCP y llama a *bind* y a *listen*.

```

static struct sockaddr_in ip_address(const char* host,
                                     const char* service,
                                     const char* proto);

static int tcp_master_socket(const char* service)
{
    struct sockaddr_in sin = ip_address("0.0.0.0", service, "tcp");
    struct protoent* pe = getprotobyname("tcp");
    assert(pe != NULL);

    int fd = socket(PF_INET, SOCK_STREAM, pe->p_proto);
    assert (fd >= 0);
    assert(bind(fd, (struct sockaddr*)&sin, sizeof(sin)) >= 0);
    assert(listen(fd, 10) >= 0);
    return fd;
}

```

Al igual que antes *bind* asigna la dirección y puerto en los que escucha, pero ahora tenemos que configurar el *backlog* con *listen* (cuántas conexiones se retienen sin aceptar) y aceptar nuevas conexiones con *accept*.

En realidad esto es una sobre-simplificación. Una vez aceptada la conexión se puede atender en un hilo independiente y aceptar nuevas conexiones en el hilo principal inmediatamente.

El cliente es prácticamente idéntico al de UDP.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <netdb.h>
#include <arpa/inet.h>

static int tcp_client_socket(const char* host, const char* service);

int main() {
    int fd = tcp_client_socket("localhost", "9999");
    for(;;) {
        char buf[1024];
        fgets(buf, sizeof(buf), stdin);
        int n = write(fd, buf, strlen(buf));
        assert(n >= 0);
        if (n == 0) break;
    }
    close(fd);
    return 0;
}
```

Y la función `tcp_client_socket` es prácticamente igual que su homólogo en UDP.

```
static struct sockaddr_in ip_address(const char* host,
                                     const char* service,
                                     const char* proto);

static int tcp_client_socket(const char* host, const char* service)
{
    struct sockaddr_in sin = ip_address(host, service, "tcp");
    struct protoent* pe = getprotobyname("tcp");
    assert(pe != NULL);
    int fd = socket(PF_INET, SOCK_STREAM, pe->p_proto);
    assert (fd >= 0);
    assert (connect(fd, (struct sockaddr*)&sin, sizeof(sin)) >= 0);
    return fd;
}
```

Arquitectura de software

La arquitectura de un programa no es más que su organización interna, los componentes en los que se divide y la relación entre ellos. Si buscas bibliografía sobre el tema verás centenares de libros que hablan de la arquitectura de software como una evolución de la artesanía a la ingeniería en el campo del software. No leerás en este manual nada sobre esto, porque sigo pensando que la programación es una labor artesanal, como el trabajo de los mejores ingenieros. Y como tal tiene que tener un componente estético muy importante. Valorar y entrenar el arte de programar es una parte importante de la profesionalización de la actividad.

Podemos escribir decenas de páginas sobre requisitos, diseño arquitectónico, diseño detallado, etc. Pero nada de esto podrá despertar el gusanillo de los *hackers* originales, los apasionados por la tecnología, que hacían de la mejora continua su propia forma de vida.

En este capítulo te ofrecemos una posible forma de organizar tu código. Ni es la única posible ni posiblemente sea la mejor. Cumple con su función de equilibrar la eficiencia con la simplicidad y la facilidad de uso por programadores principiantes. Pero si tienes sugerencias sobre posibles mejoras no te las guardes, envíanos realimentación para mejorar las próximas ediciones.

Tratamiento de errores

Antes de empezar a escribir programas relativamente complejos en C conviene hacer una pequeña reflexión sobre la gestión de errores en C. La forma habitual de manejar errores es devolver un código de error en las funciones y, dependiendo de su valor actuar de una forma u otra.

Pero ¿qué pasa si no sabemos cómo actuar ante una situación errónea? Es muy frecuente que en el momento que se produce el error no tengamos toda la información necesaria para tratarlo debidamente. Es habitual devolver otro código de error al llamador, pero esto empieza a oscurecer el código porque cada llamada a función tiene que ir en una cláusula `if`.

Lo correcto hoy en día es emplear un mecanismo soportado por todos los lenguajes de programación modernos, que se denomina *excepciones*. Pero C no tiene excepciones. Afortunadamente hay un conjunto de bibliotecas que implementan el mecanismo utilizando macros del preprocesador y dos funciones de la biblioteca estándar de C que no suelen ser muy conocidas, `setjmp` y `longjmp`.

No es propósito de este taller explicar el funcionamiento de estas funciones. Si tienes curiosidad mira la página de manual para entender su funcionamiento. Son esenciales para implementar corrutinas, o cualquier otra forma de interrumpir el flujo normal de ejecución de un programa.

En nuestros ejemplos vamos a optar por `cexcept` por su extrema simplicidad. Tiene un único archivo de cabecera (`cexcept.h`) y su uso es muy simple.

Cuando el programa tiene suficiente información para manejar posibles errores debe utilizar una construcción de este estilo:

```
Try {
    /* Código de usuario, sin 'manejo de errores */
}
Catch (ex) {
    /* Código para tratar el error notificado en ex */
}
```

En el momento en que se puede producir una situación excepcional o un error debe notificarse de esta forma:

```
if (funcion_tradicional() < 0)
    Throw ex;
```

Donde `ex` es una excepción, una estructura de datos que recoge toda la información del error para ser manejada cuando esto sea posible. El tipo de datos de `ex` es definible por el usuario.

En nuestros ejemplos utilizaremos `reactor/exception.h`, una biblioteca auxiliar en la que definimos la excepción como una estructura similar a esta:

```
typedef struct {
    int error_code;
    const char* what;
} exception;

define_exception_type(exception);
```

Así que si necesitas una descripción textual del error puedes usar el campo `what` de la estructura. Por ejemplo:

```
exception ex;
Try {
    guardar_datos(db);
}
Catch (ex) {
    printf("No pudo grabar los datos.\n"
        "Motivo: %s\n", ex.what);
}
```

Para notificar una situación excepcional se dice que se eleva una excepción. Es decir, se usa una sentencia `Throw` con los valores apropiados de la excepción. Por ejemplo:

```
f = fopen(fname, "r");
if (f == NULL)
    Throw Exception(errno, "")
```

Cuando se ejecuta la sentencia `Throw` el programa pasa el control a la cláusula `Catch` del último `Try` abierto, aunque esté en otra función. Es un mecanismo muy poderoso para desacoplar la lógica del programa y la lógica de manejo de errores.

Anidamiento de `Try/Catch`

En algunos casos se sabe cómo manejar algunos errores, pero no todos. En esos casos puede ser útil capturar la posible excepción y volverla a lanzar si no se sabe qué hacer con ella. Por ejemplo:

```
void f() {
    exception e;
    Try {
        procesamiento_complejo();
        mas_procesamiento_complejo();
        todavia_mas_procesamiento_complejo();
    }
    Catch(e) {
        if (e.error_code != ERROR_NO_MAS_DATOS)
            Throw e;
    }
}

void g() {
    exception e;
    Try {
        f();
    }
    Catch(e) {
        fprintf(stderr, "Error: %s\n", e.what());
    }
}
```

La función `f` realiza el procesamiento complejo y captura la excepción `ERROR_NO_MAS_DATOS` porque entiende que es una situación normal, que no necesita ser tratada de ninguna forma especial. Sin embargo en cualquier otro caso relanza la excepción para que se trate más arriba en la cadena de llamadas.

La función `g`, que es de mayor nivel de abstracción, utiliza `f` pero en caso de fallo lo notifica al usuario por la salida de error estándar. Fíjate cómo se reparte la responsabilidad de emitir el error (e.g. en `procesamiento_complejo`) y de tratarlo en el punto donde se sabe cómo tratarlo (`f` o `g`). No es necesario devolver códigos de error en todas las funciones, ni añadir `if` cuando no se sabe cómo actuar con el error.

Prevenir *leaks*

Las excepciones pueden ser un mecanismo muy efectivo para evitar que algunos recursos se queden sin liberar (memoria dinámica reservada con `malloc` que no se libera con `free`, archivos abiertos con `fopen` que no se cierran con `fclose`, archivos abiertos con `open` que no se cierran con `close`, etc.). Por ejemplo, considera esta función para contar el número de líneas de un archivo:

```
int contar_lineas(const char* nombre)
{
    FILE* f = fopen(nombre, "r");
    int lineas = 0;
    char buf[128];
    while (NULL != fgets(buf, sizeof(buf), f))
        if (buf[strlen(buf)-1] == '\n')
            ++lineas;
    if (buf[0] != '\0' && buf[0] != '\n')
        ++lineas;
    fclose(f);
    return lineas;
}
```


Esta función cuenta correctamente el número de líneas de un archivo contando los caracteres terminador de línea `'\n'` que aparecen y corrigiendo la cuenta si la última línea no tiene terminador de línea y no está vacía. Pero ¿qué pasa si hay error?

Podemos aprovechar lo que ahora sabemos para notificar errores:

```
int contar_lineas(const char* nombre)
{
    FILE* f = fopen(nombre, "r");
    if (f == NULL)
        Throw Exception(errno, "Error al abrir archivo");
    int lineas = 0;
    char buf[128];
    while (NULL != fgets(buf, sizeof(buf), f))
        if (buf[strlen(buf)-1] == '\n')
            ++lineas;
    if (ferror(f))
        Throw Exception(errno, "Error de lectura");
    if (buf[0] != '\0' && buf[0] != '\n')
        ++lineas;
    fclose(f);
    return lineas;
}
```

El problema es que en caso de error de lectura el archivo `f` no se cierra nunca. Nadie llama a `fclose`. La solución es poner la excepción pero no lanzarla hasta el final.

```
int contar_lineas(const char* nombre)
{
    FILE* f = fopen(nombre, "r");
    if (f == NULL)
        Throw Exception(errno, "Error al abrir archivo");
    int lineas = 0;
    char buf[128];
    while (NULL != fgets(buf, sizeof(buf), f))
        if (buf[strlen(buf)-1] == '\n')
            ++lineas;
    exception e = { -1, NULL };
    if (ferror(f))
        e = Exception(errno, "Error de lectura");
    else if (buf[0] != '\0' && buf[0] != '\n')
        ++lineas;
    fclose(f);
    if (e.error_code >= 0)
        Throw e;
    return lineas;
}
```

Para mayor generalidad, para el caso de que haya funciones anidadas que usan excepciones, o si el flujo en caso de error no está claro, es mejor meter el código en un `Try/Catch`. Por ejemplo, en el siguiente fragmento separamos el código en dos funciones, la función `contar_lineas_en_file` ni siquiera sabe que se tenga que liberar ningún recurso.

```

int contar_lineas_en_file(FILE* f)
{
    int lineas = 0;
    char buf[128];
    while (NULL != fgets(buf, sizeof(buf), f))
        if (buf[strlen(buf)-1] == '\n')
            ++lineas;
    if (ferror(f))
        Throw Exception(errno, "Error de lectura");
    if (buf[0] != '\0' && buf[0] != '\n')
        ++lineas;
    return lineas;
}

int contar_lineas(const char* nombre)
{
    int ret = 0;
    exception e = { -1, NULL };
    FILE* f = fopen(nombre, "r");
    if (f == NULL)
        Throw Exception(errno, "Error al abrir archivo");
    Try {
        ret = contar_lineas_en_file(f);
    }
    Catch(e) {
    }
    fclose(f);
    if (e.error_code >= 0)
        Throw e;
    return lineas;
}

```

Esta construcción es equivalente a lo que en Java se denomina *cláusula finally*. No importa lo que haga el programa dentro del `Try`, en cualquier caso se llamará a `close`.

Excepciones no capturadas

En una situación normal si una excepción de *cexcept* no es capturada con una sentencia `Try/Catch` se produciría un fallo de segmentación. En `reactor` hemos proporcionado un mecanismo para que las excepciones no capturadas desemboquen en un mensaje de información sobre el error y un volcado de la traza de llamada. Por ejemplo:

```

Uncaught exception (error_code 111) at socket_handler.c:167
Error en connect: Connection refused
Current call trace (last 5):
./rpi-src/c/reactor/test/test_connector(connector_init+0x28)[0x12d1c]
./rpi-src/c/reactor/test/test_connector(connector_new+0x3c)[0x12cd4]
./rpi-src/c/reactor/test/test_connector(main__+0x5c)[0x11b0c]
./rpi-src/c/reactor/test/test_connector(main+0xa0)[0x13870]
/lib/arm-linux-gnueabi/libc.so.6(__libc_start_main+0x114)[0x76d4b294]

```

Informa de que la excepción se ha producido en la línea 167 del archivo `socket_handler.c`. Para mayor información se notifica la lista de llamadas no terminadas en el momento de la excepción. En este ejemplo la excepción se produjo en la función `connector_init` que fue llamada desde `connector_new` y ésta desde `main__`. Nota que el nombre de `main` aparece algo extraño. Eso es por el mecanismo incorporado para tratar las excepciones no capturadas.

Uso en tus programas

Para usar excepciones en tus programas solo tienes que tener en cuenta algunos detalles.

- Incluye el archivo de cabecera `reactor/exception.h`.
- Compila con las opciones `-fasynchronous-unwind-tables` y `-pthread`.
- Añade al montador las opciones `-pthread` y `-rdynamic`.

Realmente solo el primer punto es estrictamente necesario, pero los demás consejos permiten tener información completa de la traza de llamadas, una gran ayuda para depurar.

Simplificar con excepciones

Separar el flujo de control del programa del flujo de control de errores tiene una consecuencia clara a la hora de escribir programas. Se fortalecen las abstracciones y es más fácil escribirlos. Veamos un ejemplo sencillo para ver esto.

Una función muy frecuente en los programas de Raspberry Pi es `write`. Se trata de una llamada al sistema operativo que escribe un conjunto de datos en un archivo identificado por un descriptor de archivo. Ese archivo puede representar un puerto serie, un canal de comunicaciones WiFi con otra Raspberry Pi, o incluso una salida digital. La esencia de Unix y todos sus derivados e imitaciones es que todo en el sistema se vea como un archivo desde el punto de vista del sistema operativo.

```
ssize_t write(int fd, const void* buf, size_t size);
```

La llamada al sistema `write` devuelve un valor que normalmente corresponde al número de bytes escritos. En general no tiene por qué coincidir con el número de bytes totales y no por ello implica error. Para poder mandar todo hay que volver a llamar a `write` con el resto. Por tanto este uso es típico:

```
int escribir_datos(int fd, const tipo_datos* datos)
{
    const void* buf = datos;
    int size = sizeof(tipo_datos);
    while (size > 0) {
        int n = write(fd, buf, size);
        if (n < 0)
            return -1;
        buf += n;
        size -= n;
    }
    return 0;
}
```

Es innecesariamente largo debido a que `write` no devuelve los bytes escritos, sino que tiene dos posibles significados. Con excepciones esto no pasa, vamos a envolver `write` en una función más amigable:

```
int write_ex(int fd, const void* buf, size_t size)
{
    if (0 > write(fd, buf, size))
        Throw Exception(errno, "write error");
}
```

No ha sido muy complicado pero hemos eliminado la fuente de la confusión. Ahora si devuelve algo es el número de bytes escritos, no devuelve ningún error. Si hay errores se trata aparte, en el `Try/Catch` correspondiente. Ahora se puede simplificar considerablemente:

```
void escribir_datos(int fd, const tipo_datos* datos)
{
    const void *buf = datos, *end = buf + sizeof(tipo_datos);
    while ((buf += write_ex(fd, buf, end - buf)) < end)
        ;
}
```

No hace falta devolver otro código de error, si hay excepción ya se propagará adecuadamente.

No te quedes en esta breve explicación, mira tranquilamente los ejemplos que te proporcionamos en el taller, especialmente la biblioteca `reactor`. Aunque las excepciones te parezcan algo completamente nuevo empieza a usarlas desde ya. Tu código ganará mucho en legibilidad y en robustez, aunque no tengas muy claro cómo funciona.

Otros contextos de excepción

En `reactor` no nos hemos preocupado demasiado del manejo de excepciones en los hilos que no son el hilo principal. En la práctica esta limitación no es tan importante, porque los hilos auxiliares son habitualmente muy simples. Cuando la cosa se complica suele usarse un *proceso* diferente, por ejemplo con un `process_handler`.

Sin embargo hemos habilitado un proceso para permitir el uso de excepciones en múltiples hilos de ejecución cambiando el denominado *contexto de excepción*. Por ejemplo, imagina que tenemos dos hilos que realizan un trabajo importante. Por ejemplo, cada uno con su reactor. El hilo principal tiene el contexto de ejecución 0 (cero). Si queremos tener excepciones en el otro hilo basta asignarle un contexto de excepción diferente al principio del hilo:

```
void* thread(thread_handler* t)
{
    current_exception_context = 1;
    ...
}
```

Es así de simple, pero hay que tener cuidado de que todos los hilos con excepciones tengan un `current_exception_context` diferente. El número de contextos disponible es reducido, pero puede cambiarse fácilmente en `reactor/exception.c` editando el valor de la constante `NUM_EXCEPTION_CONTEXTS` y recompilando.

Fundamentos de POO

POO es la abreviatura de programación orientada a objetos. Puede que te haya extrañado el título de esta sección. ¿Programación orientada a objetos? ¿En C? La programación orientada a objetos no es más que una técnica de programación. Cuando se dice que un lenguaje *soporta* POO significa que incorpora mecanismos específicos para hacer más fácil la POO. Pero en cualquier lenguaje actual se puede programar orientado a objetos.

Un objeto no es más que una zona de memoria con semántica asociada. Es decir, que tienen significado y hay un conjunto de operaciones que tiene sentido realizar sobre ellos. Un entero, por ejemplo, es un objeto en C. Tiene significado matemático y se pueden realizar las operaciones de suma, resta, etc. ¿Cómo hacemos objetos arbitrarios en C? Por ejemplo, ¿cómo hacemos un objeto que represente un `empleado` en un registro de personal?

La respuesta es con ayuda de una estructura, por ejemplo:

```
typedef struct empleado_ empleado;
struct empleado_ {
    const char* nombre;
    const char* puesto;
    double sueldo;
};
```

Los distintos elementos de la estructura se denominan *atributos del objeto*. Es todo aquello que le da significado a cada uno de los objetos.

Pero todavía no hemos hecho objetos, solo hemos definido la forma que tendrán esos objetos, es lo que se denomina *clase*. Para construir objetos se definen funciones que reciben los parámetros necesarios y devuelven un puntero a una estructura de éstas, los constructores. Por ejemplo:

```
empleado* empleado_new(const char* nombre,
                      const char* puesto,
                      double sueldo)
{
    empleado* this = (empleado*)malloc(sizeof(empleado));
    empleado_init(this, nombre, puesto, sueldo);
    return this;
}

void empleado_init(empleado* this,
                  const char* nombre,
                  const char* puesto,
                  double sueldo)
{
    this->nombre = strdup(nombre);
    this->puesto = strdup(puesto);
    this->sueldo = sueldo;
}
```

Los constructores en C suelen dividirse en dos partes, un `xxx_new` y un `xxx_init`. El primero reserva espacio en memoria dinámica, mientras que el segundo solo rellena los datos. Esto resulta útil para poder definir empleados en memoria dinámica o en variables automáticas:

```
empleado* ed = empleado_new("Paco", "Jefe", 1800.);
empleado ea;
empleado_init(&ea, "Pepe", "Currito", 1000.);
```

El empleado `ed` es un objeto en memoria dinámica, que debe ser liberado llamando a `free`. Sin embargo el empleado `ea` es un objeto en la pila, que se libera automáticamente cuando termine el ámbito de declaración. La segunda forma es también muy útil para construir vectores de `empleado`.

Algunos de vosotros os habréis dado cuenta de que aquí falla algo. Si el objeto `ea` se libera no vamos a poder liberar las cadenas correspondientes a `nombre` o `puesto`. La solución es utilizar una función especial para liberar todo lo reservado en el constructor, el destructor. Pero lo que hay que liberar es diferente si se ha utilizado `xxx_new` o `xxx_init`. ¿Cómo conseguimos que se comporte de forma diferente dependiendo de dónde haya sido construido?

La técnica para resolver este problema se denomina *funciones virtuales*. Hay varias formas de implementarlo, lo haremos de la forma más simple posible. En lugar de usar una función para liberar el objeto vamos a usar un puntero a función y ese puntero cambiará dependiendo del modo en que se ha construido el objeto. El decir, el propio objeto lleva no solo datos sino también punteros a funciones que pueden cambiar dependiendo del caso.

```
typedef struct empleado_ empleado;
typedef void (*empleado_func)(empleado*);
struct empleado_ {
    const char* nombre;
    const char* puesto;
    double sueldo;
    empleado_func destroy;
};
```

Y el destructor puede ser una función tan simple como:

```
void empleado_destroy(empleado* this)
{
    this->destroy(this);
}
```

Evidentemente el trabajo no está en esa función sino en aquella a la que apunta `e->destroy`. Y además es preciso garantizar que siempre apunta a una función válida, porque de otro modo al llamar al destructor se produciría un error catastrófico. Ésta y cualquier otra garantía que sea preciso mantener para que el estado sea siempre consistente se denominan *invariantes de clase*.

El constructor es responsable de *establecer* los invariantes de clase. Todas las demás operaciones son responsables de *mantener* los invariantes de clase. Veamos cómo quedaría la función `empleado_init`.

```
static void empleado_free_members(empleado*);

void empleado_init(empleado* this,
                  const char* nombre,
                  const char* puesto,
                  double sueldo)
{
    this->nombre = strdup(nombre);
    this->puesto = strdup(puesto);
    this->suelo = sueldo;
    this->destroy = empleado_free_members;
}

static void empleado_free_members(empleado* this)
{
    free(this->nombre);
    free(this->puesto);
}
```

Pero esto no libera la estructura de `empleado` cuando se construye con `empleado_new`. Así que en ese caso habrá que cambiar el destructor:

```
static void empleado_free(empleado*);

empleado* empleado_new(const char* nombre,
                      const char* puesto,
                      double sueldo)
{
    empleado* this = (empleado*)malloc(sizeof(empleado));
    empleado_init(this, nombre, puesto, sueldo);
    this->destroy = empleado_free;
    return this;
}

static void empleado_free(empleado* this)
{
    empleado_free_members(this);
    free(this);
}
```

La función `empleado_destroy` es un ejemplo de operación que se puede realizar sobre un `empleado`. Estas operaciones se llaman de forma general *métodos* del objeto, y por tratarse de la operación que libera los recursos del `empleado` se llamaría de forma más específica *destructor*. Además es un método que se comporta de forma diferente según el tipo de `empleado` sobre el que se llame. Este tipo de *métodos* se llaman en general *métodos virtuales*, y al tratarse de un destructor sería también de forma más específica *destructor virtual*.

Los *métodos virtuales* se utilizan en combinación con otra característica muy importante de la POO, la herencia. En el ejemplo de `empleado` podemos considerar el caso de un `gerente` que básicamente funciona igual que un `empleado`, pero tiene otras características, como por ejemplo bonus por productividad. Por tanto el cálculo de las retribuciones anuales será diferente según se trate de un empleado normal o de un gerente. Eso se puede conseguir añadiendo otra función virtual para ello.

```
typedef struct empleado_ empleado;
typedef void (*empleado_func)(empleado*);
typedef double (*empleado_func_double)(empleado*);
struct empleado_ {
    const char* nombre;
    const char* puesto;
    double sueldo;
    empleado_func destroy;
    empleado_func_double retribuciones;
};
```

El campo `retribuciones` se inicializa en el constructor, de forma similar a `destroy` utilizando la siguiente función:

```
static double empleado_retribuciones_14pagas(empleado* this)
{
    return 14.*this->sueldo;
}
```

Con lo que el método quedaría tan simple como:

```
double empleado_retribuciones(empleado* this)
{
    return this->retribuciones(this);
}
```

Sin embargo, en el caso del gerente tendríamos un objeto con más atributos:

```
typedef struct gerente_ gerente;
typedef void (*gerente_func)(gerente*);
struct gerente_ {
    empleado parent;
    double bonus;
};
```

Hemos colocado como primer elemento de la estructura un `empleado`. Esto hace que la primera parte de un `gerente` sea la que corresponde a un `empleado`. Por tanto los métodos de `empleado` siguen funcionando sobre un objeto de la clase `gerente`, porque solo acceden a esa primera parte.

Por otro lado, la clase `gerente` debe *redefinir* el método de cálculo de retribuciones:

```

static double gerente_retribuciones(empleado* this)
{
    gerente* g = (gerente*) this;
    double r = 14. * this->sueldo;
    if (gerente_objetivos_conseguidos(g))
        r += g->bonus;
    return r;
}

void gerente_init(gerente* this,
                  const char* nombre,
                  const char* puesto,
                  double sueldo,
                  double bonus)
{
    empleado* parent = &this->parent;
    empleado_init(parent, nombre, puesto, sueldo);
    parent->retribuciones = gerente_retribuciones;
}

```

Observa cómo la nueva implementación del método `retribuciones` convierte su argumento en un `gerente`. Si se ha llamado a este método es seguro que se trata de un gerente. Ahora podemos calcular las retribuciones de cualquier empleado sea o no gerente:

```

empleado* equipo[] = {
    (empleado*) gerente_new("Paco", "Jefe", 1800., 4000.),
    empleado_new("Pepe", "Currito", 1000.),
    empleado_new("Juan", "Currito", 1000.),
};

#define NELEMS(a) (sizeof(a)/sizeof(a[0]))
double coste_personal = 0.;
for (int i=0; i<NELEMS(equipo); ++i)
    coste_personal += empleado_retribuciones(equipo[i]);

```

La herencia es un mecanismo muy efectivo para tratar de forma homogénea a objetos, pero introduce muchísimo acoplamiento. Intenta minimizarla todo lo posible.

Esta implementación de POO en C es primitiva pero para los fines del taller nos bastará. En proyectos reales convendría echar un vistazo a las bibliotecas que ya existen. En particular merece la pena destacar [GObject](#) y [COS](#). Cada una tiene sus ventajas y sus inconvenientes que habrá que valorar.

La biblioteca Reactor

Este capítulo tiene dos objetivos:

- Ofrecer unas pinceladas de los patrones de diseño que consideramos más importantes para el desarrollo con Raspberry Pi. Documentar brevemente los fundamentos y cómo se aplican.
- Proporcionar plantillas de aplicación totalmente funcionales y extensibles para aplicarlas en cualquier proyecto. En un capítulo posterior se desarrollarán en diversos casos de estudio, incluyendo la interacción a través de la red, la interacción con programas externos, etc.

El patrón reactor

Uno de los mayores beneficios de utilizar programación orientada a objetos es que nos abre la posibilidad de utilizar directamente la mayor parte del catálogo de *patrones de diseño* disponibles en la actualidad. Un *patrón de diseño* es una solución de diseño bien probada a un problema o conjuntos de problemas y suele describirse de una manera semiformal en términos de objetos y relaciones entre ellos. El primer libro publicado sobre este tema (se conoce popularmente como *Gang of Four*, GOF) sigue siendo una referencia fundamental [Citation not found] pero ya no es la única. Cabe destacar la serie de volúmenes de *Pattern-Oriented Software Architecture* (POSA) [Citation not found] [Citation not found] [Citation not found] [Citation not found] [Citation not found].

Para este taller es especialmente importante el patrón *reactor* que se encuentra descrito en el volumen 2 de POSA. Se trata de un patrón arquitectural, en el sentido de que determina la estructura de la aplicación en términos de componentes y cómo se relacionan. Nos ayuda a organizar el programa cuando se trata de un *sistema reactivo*. Es decir, cuando el sistema responde lo más rápidamente posible a *eventos* externos o internos.

Un sistema electrónico es muy frecuentemente un sistema reactivo. Reacciona cuando se pulsan botones, o se detecta luz, o se detecta proximidad, o se detecta humedad, o se recibe un flanco de la señal de un *encoder*, o se activa un *final de carrera*, o se pulsa una tecla del teclado, o se pulsa un botón del ratón, o se recibe un dato por la red, ... Prácticamente todo lo que proporciona datos al sistema (entradas) puede modelarse como fuentes de eventos.

Cualquier desarrollo progresará de forma incremental, primero con una fuente de eventos, y luego otra, y luego otra. No es infrecuente ver programas que se convierten en un auténtico galimatías que es imposible hacer funcionar. La gama de aberraciones es muy amplia:

- Algunos utilizan un bucle infinito enorme, en el que se leen todas las entradas y se van haciendo cálculos parciales mezclados con el proceso de lectura de eventos. Frecuentemente no es posible tener un resultado hasta que vengan otros eventos, así que se guardan valores en variables temporales para tratarlo en otras iteraciones del bucle. La maraña de `if` encadenados se hace inmanejable. El resultado es extremadamente difícil de seguir hasta para un experto. Solo pensar en añadir una nueva entrada produce escalofríos.
- Otros hacen todo en manejadores de interrupción. Al pulsar un botón se dispara un manejador de interrupción que se utiliza directamente para cambiar el color de un LED. Cuando el sistema es diminuto parece que funciona de maravilla. Luego va creciendo y empiezan los problemas. Primero fallos catastróficos inexplicables, que terminan en maldiciones contra los punteros de C. Luego, cuando se es consciente de la necesidad de *async-safety* se empiezan a utilizar primitivas de sincronización y aparecen los interbloqueos.

No hay nada que podamos hacer para arreglar esto, la única solución rentable a largo plazo es analizar el programa para ver cada requisito funcional y volverlo a construir desde cero con una arquitectura razonable. Hay gente que no cree esto e invierte meses en intentar depurar el software.

Incluso he llegado a ver casos en los que aparentemente se ha conseguido que funcione, pero es una falacia, no es real. ¿Por qué? Pues simplemente porque se equivoca el propósito del programa. El programa no está solo ni principalmente para ser ejecutado. Está fundamentalmente para ser leído y modificado. Por eso usamos un lenguaje de alto nivel. La vida del programa no acaba cuando se ejecuta.

Una arquitectura incorrecta pone una losa en la espalda de todos los que tengan que modificar el programa en el futuro. Es peor que no hacer nada, es comprometer gasto en el futuro. La productividad de estos programadores habría que contabilizarla como un número negativo.

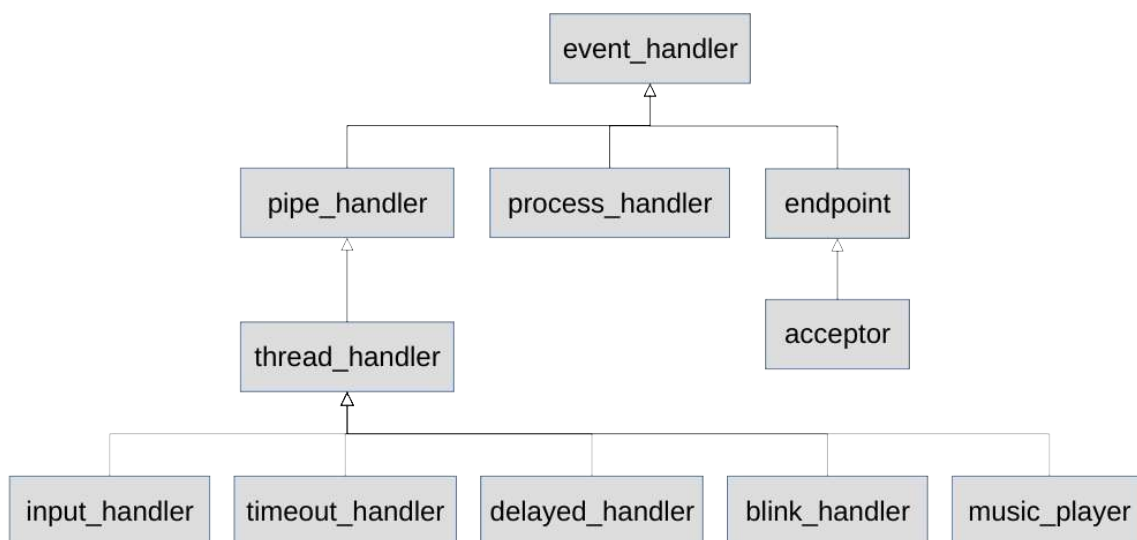
El patrón *reactor* no es la única arquitectura posible, pero es probablemente la más apropiada para el tipo de sistemas de este taller. Consiste en asociar cada tipo de eventos con un objeto manejador (`event_handler`). El manejador tiene un método para consumir los eventos pendientes de ese tipo (`handle`). Por otro lado hay un objeto `reactor` que encapsula todo lo que tiene que ver con la demultiplexación de eventos (*dispatching*). Cuando ocurre cierto evento busca al manejador correspondiente para invocar su método `handle_events`.

En el taller utilizaremos una implementación propia de este patrón que se incluye en la carpeta `src/c/reactor`. Incluye numerosas implementaciones de diferentes `event_handler` para eventos de interés. Tómate tu tiempo para analizarlos en detalle.

Cada vez que incorporemos una nueva fuente de eventos es necesario crear un manejador de eventos asociado y registrarlo en el `reactor`. También puede considerarse el tiempo como una fuente de eventos que se tratan con diversos manejadores especializados.

El `reactor` tiene un método `reactor_run` que implementa el denominado *bucle de eventos*. Es el bucle en el que se detecta si hay eventos disponibles y en caso de que los hubiera se *despachan* usando su manejador correspondiente.

Vamos a hacer un breve recorrido por los manejadores de eventos incluidos en la biblioteca `reactor` del taller.



Jerarquía de manejadores incluidos actualmente en la biblioteca reactor.

Eventos de teclado

El teclado en C suele leerse con funciones como `getchar`. Pero ésta, como todas las demás funciones de `stdio`, no produce un valor inmediatamente cuando se pulsa una tecla. Se puede pulsar toda una secuencia de teclas pero hasta que no se presione la tecla *Intro* no se recibirá ni una sola pulsación.

Este modo de funcionamiento está pensado para aliviar la carga de trabajo en el modo habitual de uso, para editar textos u órdenes. Los programas solo reciben información cuando tienen algo significativo que hacer.

Las funciones de `stdio` utilizan siempre objetos de tipo `FILE`. Incluso cuando no se indica (`printf`, `scanf`) trabajan con variables `FILE` globales (`stdout`, `stdin`). Estos objetos incorporan *arrays* que actúan como *buffers* de la entrada. Retienen los caracteres leídos hasta que se disponga de suficiente información. Evidentemente con `FILE` nunca vamos a conseguir una respuesta inmediata al pulsar una tecla.

Tenemos que mirar por tanto a la interfaz del *sistema operativo*, lo que usa `stdio` en su implementación. Y aquí estamos de enhorabuena, porque GNU replica el modelo de Unix que se caracteriza por su simplicidad.

- En Unix todo son archivos o dispositivos que se comportan como archivos (teclado, ratón, terminales, puertos serie, discos, micrófono, red, gráficos, etc.)
- Todos los archivos y dispositivos se manejan con cuatro operaciones básicas: `open`, `read`, `write`, `close` y excepcionalmente con una quinta `ioctl`.

- Para usar un archivo o dispositivo debe abrirse con `open`. A partir de entonces se obtiene un entero (*descriptor de archivo*) que representa el archivo en todas las demás llamadas.
- En todos los programas la entrada estándar está disponible como un archivo ya abierto con el *descriptor de archivo* 0. Análogamente la salida estándar y la salida de error estándar están disponibles en los descriptores 1 y 2 respectivamente.

Esta uniformidad de Unix es lo que explota nuestra implementación del *reactor* para que la demultiplexión de eventos sea siempre de archivos Unix. Ya veremos qué hacer cuando no hay tales archivos (e.g. GPIO).

Visto esto parece que lo único que tenemos que hacer es leer del descriptor 0 usando `read`. Pero si lo intentamos veremos que sigue sin recibirse las teclas hasta que se pulse *Intro*. El motivo es que el dispositivo está configurado para que solo interrumpa al procesador si tiene pendientes cierto número de caracteres, o si ha pasado cierto tiempo desde la última interrupción, o si se ha pulsado *Intro*.

En los archivos `console.h` y `console.c` de la biblioteca `reactor` te proporcionamos un par de funciones `console_set_raw_mode` y `console_restore`. La primera permite la realimentación inmediata del teclado, configurando los parámetros adecuados y devuelve un *puntero opaco* con información de la configuración previa. La segunda permite restaurar la configuración previa. Veamos un ejemplo de uso en combinación con el `reactor`.

Siguiendo el patrón *reactor* encapsulamos la interacción con cualquier fuente de eventos como un *event handler*:

```
#include <reactor/reactor.h>

static void keyboard(event_handler* ev);

event_handler* keyboard_handler_new() {
    return event_handler_new(0, keyboard);
}

static void keyboard(event_handler* ev) {
    char buf[1];
    if (read(ev->fd, buf, 1) < 0 || buf[0] == 'q')
        reactor_quit(ev->r);
    printf("Pulsado %c\n", buf[0]);
}
```

Este manejador de ejemplo utiliza al `reactor` porque invoca a su método `reactor_quit` (salir del bucle de eventos del *reactor*) cuando detecta una condición de terminación. Para ello utiliza uno de los atributos del *event handler*. Fíjate en la llamada a `event_handler_new`. Delegamos en `event_handler` todo lo que podemos y le pasamos el descriptor de archivo 0 y la función de manejo de eventos. El descriptor 0 corresponde a la entrada estándar.

El programa principal es sencillo:

```
#include <reactor/reactor.h>
#include <reactor/console.h>

int main()
{
    void* state = console_set_raw_mode(0);
    reactor* r = reactor_new();
    reactor_add(r, keyboard_handler_new());
    reactor_run(r);
    console_restore(0, state);
    return 0;
}
```

Primero ponemos la entrada estándar (descriptor 0) en modo *raw* para tener respuesta inmediata de las pulsaciones de tecla. Luego construimos el *reactor* y todos sus manejadores de eventos (en este caso solo el del teclado). Registramos los manejadores en el *reactor* y entramos en el bucle de eventos (`reactor_run`). Una vez terminado el programa dejamos la consola en el estado normal.

Algunos de vosotros estaréis pensando que esto es matar moscas a cañonazos, que es mucho más simple un bucle sin más, sin reactor ni manejadores. Algo de este estilo:

```
int main()
{
    void* state = console_set_raw_mode(0);
    for(;;) {
        char buf[1];
        if (read(0, buf, 1) < 0 || buf[0] == 'q')
            break;
        printf("Pulsado %c\n", buf[0]);
    }
    console_restore(0, state);
}
```

Funciona exactamente igual, es mucho más corto y hasta se entiende mejor. ¿Por qué complicarlo? Pues porque la vida es compleja, no es así de simple. No hay una fuente de eventos, hay decenas. No hay que procesar un evento, sino varios. No son eventos independientes, sino relacionados, hay eventos periódicos, hay fuentes que aparecen y desaparecen, etc. Te proponemos un ejercicio, empieza con este ejemplo y sigue con él haciendo en paralelo todos los ejemplos que te mostramos. Después si lo consigues comparamos.

Si te frustra escribir tanto cambia de lenguaje. El problema no está en el programa, sino en el lenguaje, que es demasiado primitivo. Usa C++ o Python.

Eventos en una entrada digital

Las patas de GPIO configuradas como entradas son una fuente frecuente de eventos para el sistema. De hecho en un sistema empotrado será mucho más frecuente que eventos de un teclado USB. Los teclados de los sistemas empotrados se implementan frecuentemente con patas de GPIO.

Para tratar estos eventos en cualquier conjunto de entradas incluimos el `input_handler`. Se configuran automáticamente como entradas con *pull down* y se detectan tanto las transiciones de nivel bajo a alto (*press*) como de nivel alto a bajo (*release*).

```
#include <reactor/reactor.h>
#include <reactor/input_handler.h>
#include <wiringPi.h>
#include <stdio.h>

static void press(input_handler* ev, int key) { printf("Press %d\n", key); }
static void release(input_handler* ev, int key) { printf("Release %d\n", key); }

int main()
{
    int buttons[] = { 18, 23, 24, 25 };

    wiringPiSetupGpio();
    reactor* r = reactor_new();
    reactor_add(r, (event_handler*) input_handler_new(buttons, 4,
                                                       press, release));

    reactor_run(r);
}
```

Este ejemplo muestra cómo configuraríamos un conjunto de cuatro botones en la Raspberry Pi. Se detectarían tanto las pulsaciones como las liberaciones de cada botón.

Eventos disparados por tiempo

Además de los eventos que ocurren en el entorno, hay otra magnitud de importancia capital en los sistemas reactivos, el tiempo. Hay multitud de características que dependen del tiempo. Por ejemplo, un simple LED puede parpadear durante unos segundos. Esto exige contar el tiempo en que el LED se enciende o se apaga y también el tiempo total de parpadeo.

Eventos periódicos

Nuestra implementación del *reactor* implementa *periodic handlers*. Un *periodic handler* crea un nuevo hilo que va generando eventos cada cierto número de milisegundos en un descriptor especial denominado *pipe* (tubería). Realmente los detalles no es necesario conocerlos, pero conviene saber que aunque aparentemente el tiempo no está relacionado con un archivo nuestra implementación lo traduce a eventos en un archivo especial.

Usar *periodic handlers* es igual de sencillo que cualquier otro:

```
#include <reactor/reactor.h>
#include <reactor/periodic_handler.h>
#include <stdio.h>

void handler(event_handler* ev)
{
    puts("Tick");
}

int main()
{
    reactor* r = reactor_new();
    reactor_add(r, (event_handler*)periodic_handler_new(1000, handler));
    reactor_run(r);
    return 0;
}
```

Este ejemplo imprime un mensaje cada 1000 milisegundos, es decir, cada segundo. El tiempo no es absolutamente preciso. En GNU/Linux no es posible

Los manejadores de eventos se desinstalan automáticamente cuando ocurre una excepción, así que si queremos ejecutar un evento cierto número de veces podemos hacerlo contando el número de disparos.

```
void handler(event_handler* ev)
{
    static int i = 0;
    puts("Tick");
    if (++i >= 5)
        Throw Exception(0, "");
}

int main()
{
    reactor* r = reactor_new();
    reactor_add(r, (event_handler*)periodic_handler_new(100, handler));
    reactor_run(r);
    return 0;
}
```

Warning Este ejemplo dispara el periodic cinco veces. ¿Qué pasaría si pasado cierto tiempo añadimos otra vez al reactor otro *periodic handler* igual? ¿Funcionaría? Propón una solución y discútela con tus compañeros.

Eventos retardados

Otro tipo de eventos disparados por tiempo es el caso de los eventos retardados. Por ejemplo, pasado cierto tiempo si no se cumple cierta condición salir del programa.

```
#include <reactor/reactor.h>
#include <reactor/delayed_handler.h>

void timeout(event_handler* ev)
{
    if (condicion_de_salida(ev))
        reactor_quit(ev->r);
}

int main()
{
    reactor* r = reactor_new();
    ...
    reactor_add(r, (event_handler*) delayed_handler_new(1500, timeout));
    reactor_run(r);
}
```

Un `delayed_handler` precisamente hace esto. Espera cierto tiempo de milisegundos antes de invocar el manejador. Una vez invocado se desinstala automáticamente.

Parpadeo de un LED

Un caso de evento temporal que es muy frecuente en un sistema empotrado es el de hacer parpadear un LED cierto número de veces. Esto se consigue con un manejador especial denominado `blink_handler`. Se indica el pin donde se conecta, el número de milisegundos que debe permanecer en cada estado (encendido o apagado) y el número de parpadeos (ciclos encendido/apagado) que debe realizar. Se encarga automáticamente de configurar como salida el pin y de destruir el manejador una vez terminada la secuencia.

```
#include <reactor/reactor.h>
#include <reactor/blink_handler.h>
#include <wiringPi.h>

int main()
{
    wiringPiSetupGpio();
    reactor* r = reactor_new();
    reactor_add(r, (event_handler*) blink_handler_new(18, 200, 5));
    reactor_run(r);
}
```

En este caso el LED conectado a la pata GPIO18 parpadeará 5 veces con un periodo de 400ms (200ms encendido y 200ms apagado).

Programación en red

Ya hemos visto la interfaz de programación *socket* que ofrece GNU/Linux para programar comunicaciones en red. Solo nos falta organizarlo de manera razonable. Hemos visto que en las comunicaciones se pueden identificar dos roles, el rol del *servidor* y el rol del *cliente*. El rol del *servidor* es pasivo, espera hasta que ocurra un evento (la conexión o el envío de datos) y entonces reacciona.

Para ello *Reactor* proporciona un manejador denominado *endpoint* que abstrae cualquiera de los extremos de la comunicación. La forma más sencilla de comunicación corresponde a los protocolos orientados a *datagramas* como UDP. El intercambio de información se limita al envío de mensajes sin ningún tipo de confirmación. Para este caso nos basta utilizar un `endpoint`, que podemos crear con `udp_endpoint_new` para el lado del servidor o con `udp_connector_new` para el lado del cliente.

Veamos un ejemplo sencillo, un servidor de *echo*. Se trata de un programa que devuelve lo mismo que se le envía.

```
#include <reactor/reactor.h>
#include <reactor/socket_handler.h>
#include <stdio.h>

void handler(event_handler* ev)
{
    endpoint* ep = (endpoint*)ev;
    char buf[128];
    size_t len = endpoint_recv(ep, buf, sizeof(buf));
    endpoint_send(ep, buf, len);
}

int main() {
    reactor* r = reactor_new();
    reactor_add(r, (event_handler*)udp_endpoint_new("8888", handler));
    reactor_run(r);
    reactor_destroy(r);
    return 0;
}
```

Un cliente es sumamente sencillo. Simplemente utiliza un `endpoint` construido con `udp_connector_new`:

```
#include <reactor/socket_handler.h>

int main() {
    endpoint* ep = udp_connector_new("localhost", "8888", NULL);
    char buf[] = "Prueba";
    endpoint_send(ep, buf, sizeof(buf));
    endpoint_destroy(ep);
    return 0;
}
```

Un `endpoint` puede usarse también para recibir datos como un manejador más. Más adelante veremos un ejemplo.

Los protocolos orientados a conexión son algo más complejos. Implementan la ilusión de un flujo continuo de datos, lo que implica tratar de forma transparente multitud de problemas de la red. Este modelo de comunicación se divide en dos fases. Se realiza un intercambio de mensajes inicial para establecer la *conexión* única entre cliente y servidor y posteriormente se envían los datos correspondientes a esa conexión.

Acceptor-Connector es un patrón de diseño propuesto por Douglas C. Schmidt [Citation not found] y utilizado extensivamente en ACE (*Adaptive Communications Engine*), su biblioteca de comunicaciones. Se ocupa de la primera parte de la comunicación, desacopla el establecimiento de conexión y la inicialización del servicio del procesamiento que se realiza una vez que el servicio está inicializado. Para ello intervienen tres componentes: *acceptor*, *connector* y manejadores de servicio o *service handlers*. Un *connector* representa el rol activo, y solicita una conexión a un *acceptor*, que representa el rol pasivo. Cuando la conexión se establece ambos crean un manejador de servicio que procesa los datos intercambiados en la conexión.

Veamos un ejemplo sencillo, un servidor de *echo*. Se trata de un programa que devuelve lo mismo que se le envía por cualquier conexión.

```
#include <reactor/reactor.h>
#include <reactor/socket_handler.h>

void handle_echo(event_handler* ev)
{
    endpoint* ep = (endpoint*)ev;
    char buf[128];
    int n = endpoint_recv(ep, buf, sizeof(buf));
    endpoint_send(ep, buf, n);
}

int main()
{
    reactor* r = reactor_new();
    reactor_add(r, (event_handler*)acceptor_new ("10000", handle_echo));
    reactor_run(r);
    return 0;
}
```

Aparentemente es igual que el caso anterior. Se diferencian internamente y especialmente en las garantías que ofrece. Un servidor UDP puede perder mensajes o incluso recibirlos replicados. Un servidor TCP garantiza la entrega si no hay un problema físico.

El `acceptor` se encargará de llamar a *listen* y *accept* cuando sea preciso. En el momento en que se establezca una nueva conexión se creará un `endpoint` que actúa de *service handler* con el *socket* esclavo y con la función de procesamiento que se le indica. En este caso escucha en el puerto TCP 10000.

El `conector` es similar, salvo por el hecho de que en el constructor establece la conexión con el otro extremo. Si éste no está disponible se elevará una excepción. Si solo se pretende usarlo para enviar datos al servidor ni siquiera es necesario añadirlo al *reactor*.

```

#include <reactor/reactor.h>
#include <reactor/socket_handler.h>

void handler(event_handler* ev)
{
    char buf[128];
    int n = event_handler_recv(ev, buf, sizeof(buf));
    buf[n] = '\0';
    printf("Got: %s\n", buf);
}

int main()
{
    reactor* r = reactor_new();
    connector* c = connector_new("localhost", "10000", handler);
    reactor_add(r, (event_handler*)c);
    connector c_aux = *c;

    void producer(event_handler* ev)
    {
        static int i;
        char buf[128];
        snprintf(buf, 128, "Prueba %d", i++);
        connector_send(&c_aux, buf, strlen(buf));
    }

    reactor_add(r, (event_handler*)periodic_handler_new(1000, producer));
    reactor_run(r);
    return 0;
}

```

El `connector` crea el *event handler* para atender la conexión. Podemos usarlo directamente con `connector_send` y `connector_recv` o bien añadirlo a un `reactor` para responder de forma automática. En este ejemplo creamos un evento periódico que envía por el conector un mensaje distinto cada vez y añadimos el `connector` al *reactor* para recibir el mensaje de eco del servidor anterior.

Nota que el evento periódico no usa el puntero al conector directamente. En su lugar hace una copia de todo el objeto y usa esa copia. El motivo es simple, la memoria del objeto conector puede ser liberada en cualquier momento (por ejemplo, si se pierde la conexión). En ese caso ese espacio de memoria será ocupado por otros objetos y el campo correspondiente al descriptor de archivo (el socket en este caso) podría ser alterado. El evento periódico necesita enviar al descriptor correcto, aunque haya sido cerrado por un problema de comunicaciones. Si está cerrado se elevará una excepción y el evento periódico también se desinstalará automáticamente, como cabría esperar.

Este ejemplo funcionaría también con un servidor UDP cambiando el `connector` por un `udp_connector`.

Los sistemas en los que un componente actúa fundamentalmente con el rol de servidor, mientras que los otros componentes actúan como clientes siguen la *arquitectura cliente-servidor*. Es la típica en la *World-Wide-Web*.

Otras aplicaciones no tienen una división tan marcada. En un momento dado una aplicación que normalmente se comporta como servidora puede comportarse como cliente y al revés. Este tipo de sistemas en que todos los componentes toman rol de cliente o de servidor indistintamente se denominan arquitecturas *peer-to-peer*.

Un *music player* como manejador

Un sistema electrónico puede necesitar emitir sonidos. Con la Raspberry Pi, y especialmente con las últimas versiones, tenemos capacidades de sonido bastante sofisticadas y sería una pena si nos quedamos en simples *beeps*.

Actualmente la forma más habitual de guardar sonido o música de calidad es emplear el formato MP3, que es relativamente complejo. Pero no hay necesidad de decodificar los archivos en nuestros programas. Tenemos decenas de programas que hacen precisamente eso. Solo hay que usarlos, desde nuestros propios programas.

Una forma de usar otro programa desde nuestro programa es emplear un manejador especial denominado `process_handler`. Hablaremos de esto más adelante. Otra forma es utilizar un manejador a medida para controlarlo. Es este último método el que ilustra el `music_player` de la biblioteca *reactor*. Se trata de un manejador de eventos que funciona como un *music player* empleando para ello el programa `mpg123`. Para no interferir en el hilo principal la mayor parte del trabajo se deja en un hilo auxiliar que se encarga de ejecutar `mpg123` cuando es necesario, pero todo eso es invisible para el usuario. Veamos un ejemplo:


```

#include <reactor/reactor.h>
#include <reactor/console.h>
#include <reactor/music_player.h>
#include <unistd.h>

static int read_key(int fd);

int main(int argc, char* argv[])
{
    const char* carpeta = "/usr/share/scratch/Media/Sounds/Music Loops";
    void* state = console_set_raw_mode(0);
    reactor* r = reactor_new();
    music_player* mp = music_player_new(carpeta);

    void keyboard(event_handler* ev) {
        int key = read_key(ev->fd);
        if ('q' == key)
            reactor_quit(r);
        else if (' ' == key)
            music_player_stop(mp);
        else if (key >= '0' && key <= '9')
            music_player_play(mp, key - '0');
    }

    reactor_add(r, (event_handler*)mp);
    reactor_add(r, event_handler_new(0, keyboard));
    reactor_run(r);
    reactor_destroy(r);
    console_restore(0, state);
}

static int read_key(int fd)
{
    char buf[2];
    if (0 < read(fd, buf, 1))
        return buf[0];
    return -1;
}

```

En este ejemplo combinamos un manejador para el teclado con un manejador `music_player`. Al pulsar alguno de los números de 0 a 9 interrumpirá la canción en curso y sonará la canción correspondiente, al pulsar espacio parará y al pulsar `q` saldrá del programa.

Info Debes acostumbrarte a combinar manejadores. Te proponemos el siguiente ejercicio: construir un *music player* tipo iPod con la Raspberry Pi. Usa botones para ir a la siguiente canción o la anterior, pausar, etc. Se trata de combinar un `input_handler` con el `music_player`.

Otros manejadores

La biblioteca *reactor* evoluciona. Examina el código porque muy probablemente verás manejadores nuevos que no hemos descrito. Experimenta con ellos y no tengas miedo de definir tus propios manejadores cada vez que surja una necesidad.

En esta sección describiremos algunos manejadores incluidos en *reactor* que no están pensados para ser usados por el usuario final, sino por otros manejadores.

Pipe handler

Cuando queremos traducir eventos de otro tipo para que sean demultiplexados por el *reactor* se puede utilizar un `pipe_handler`. Una *pipe* es un objeto especial del sistema operativo que relaciona una pareja de descriptores de archivo. Todo lo que se escribe en un descriptor (el extremo de escritura) se puede leer por el otro (extremo de lectura).

Por ejemplo, imagina que quieres traducir determinadas interrupciones en eventos del *reactor*. Basta escribir desde la propia rutina de servicio a interrupción con `pipe_handler_write`. Si estás en un hilo que no tiene contexto de excepciones propio te interesará utilizar en su lugar `pipe_handler_write_ne`, que no eleva excepciones sino que devuelve un código de error.

Otro ejemplo podría ser un dispositivo I2C o SPI. Puede que el dispositivo interrumpa directamente o puede que haya que hacer consultas periódicas. En cualquier caso el mecanismo para detectar los eventos es distinto a un archivo Unix, por lo que necesitaremos un `pipe_handler` para adaptar.

Muchos de los manejadores que hemos visto están hechos con un `pipe_handler`. Los `input_handler`, `music_handler`, `periodic_handler`, `delayed_handler` o `blink_handler` son ejemplos de manejadores basados en `pipe_handler`.

Thread handler

En muchas ocasiones un `pipe_handler` necesita de la cooperación de un hilo de ejecución independiente, que muestrea el dispositivo o realiza algún otro tipo de acción para traducir eventos como el tiempo a escrituras en una *pipe*.

Manejar hilos distintos no es complicado pero tiene sus detalles, especialmente en cuanto a la terminación de hilos que no se comportan adecuadamente. Para simplificarlo se proporciona un `thread_handler` que deriva de `pipe_handler`. Todos los manejadores que actualmente derivan de `pipe_handler` son también `thread_handler`. Esto parece sugerir que `pipe_handler` no es realmente necesaria. Sin embargo creemos que es interesante para cualquier manejador que traduce directamente interrupciones a eventos del *reactor*.

Te preguntarás entonces por qué no lo hemos utilizado. La respuesta tiene que ver con la biblioteca *wiringPi*. Esta biblioteca tiene soporte de detección de cambios en una entrada digital mediante interrupciones. Sin embargo la API es demasiado primitiva, no hay forma de discriminar la pata que ha generado la interrupción a menos que se utilicen distintas rutinas de servicio para cada pata. De momento eso implica que necesitamos implementar un mecanismo de barrido aún cuando se activen las interrupciones. Es algo que probablemente cambiará con el tiempo.

Veamos un ejemplo sencillo de `thread_handler`:

```
#include <reactor/reactor.h>
#include <reactor/thread_handler.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <errno.h>

static void* productor(thread_handler* h) {
    struct timespec t = { .tv_sec = 0, .tv_nsec = 500000000 };
    for(int i = 1; !h->cancel; ++i) {
        pipe_handler_write_ne(&h->parent, &i, sizeof(i));
        nanosleep(&t, NULL);
    }
    return NULL;
}

static void consumidor(event_handler* ev)
{
    int n;
    pipe_handler_read((pipe_handler*)ev, &n, sizeof(n));
    printf("Got %d\n", n);
    if (n > 9)
        reactor_quit(ev->r);
}

int main()
{
    reactor* r = reactor_new();
    thread_handler* p = thread_handler_new(consumidor, productor);
    reactor_add(r, (event_handler*)p);
    reactor_run(r);
    reactor_destroy(r);
}
```

Para evitar problemas los argumentos del constructor son de tipo diferente. El primer argumento es una función de manejo de eventos normal, porque está previsto que se use como cualquier otro manejador. La segunda es el hilo por lo que tiene una signatura ligeramente diferente, compatible con la que corresponde en las funciones de creación de hilos del sistema operativo.

Process handler

Los programas con hilos son relativamente complejos de depurar porque un comportamiento inadecuado en un hilo puede afectar al comportamiento del resto de los hilos. Es muy recomendable para los programadores noveles reducir el uso de múltiples hilos a casos relativamente simples y con escasa interacción entre hilos. El `thread_handler` al estar basado sobre el `pipe_handler` implementa un mecanismo de comunicación completamente ajeno a los problemas de sincronización entre hilos, por lo que los hilos pueden ser prácticamente independientes.

Cuando el trabajo se complica también aparecen nuevas necesidades de sincronización y con ello es frecuente que aparezcan problemas de interbloqueo y *condiciones de carrera*. Depurar este tipo de problemas puede llegar a ser extremadamente complejo.

Cuando la funcionalidad de los hilos es compleja puede que interese desacoplar el sistema en procesos independientes que interactúan mediante procedimientos de comunicación de bajo acoplamiento (pipes, memoria compartida, archivos, etc.). Los procesos garantizan un mayor nivel de aislamiento que los hilos y esto contribuye a que sean más fáciles de depurar.

Un ejemplo de esta aproximación es Google Chrome. Cada pestaña del navegador se ejecuta como un proceso independiente por lo que un posible fallo en una pestaña no puede afectar a las demás. Google prima de esta forma la estabilidad frente a la eficiencia.

Otra razón para usar procesos en lugar de hilos es la necesidad de interactuar con programas externos. En ese caso las llamadas al sistema para la ejecución de programas reemplazarán por completo el proceso actual, por lo que no cabe plantear la opción de los hilos.

La biblioteca *reactor* proporciona un `process_handler` similar al `thread_handler` pero con dos pipes configuradas para la comunicación bidireccional entre ambos procesos. Ambos procesos verán un `process_handler` diferente y configurado para hablar con el otro extremo. Tienen el atributo `fd` para leer datos provenientes del otro extremo y un nuevo atributo `out` para enviar datos al otro extremo. Cada extremo es libre de añadir su vista del `process_handler` a un `reactor` o directamente ejecutar un programa externo. Veamos un ejemplo:

```
#include <reactor/reactor.h>
#include <reactor/process_handler.h>
#include <reactor/delayed_handler.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void parent (event_handler* ev) {
    process_handler* h = (process_handler*) ev;
    char buf[128];
    int n = read(ev->fd, buf, sizeof(buf));
    if (n > 0)
        printf("Padre: %s\n", buf);
    write(h->out, "padre", 6);
}

void child (event_handler* ev) {
    process_handler* h = (process_handler*) ev;
    char buf[128];
    int n = read(ev->fd, buf, sizeof(buf));
    if (n > 0)
        printf("Hijo: %s\n", buf);
    write(h->out, "hijo", 5);
}

void quit(event_handler* ev) { reactor_quit(ev->r); }

int main() {
    reactor* r = reactor_new();
    process_handler* h = process_handler_new(parent, child);
    process_handler_stay_forever_on_child(h);
    reactor_add(r, &h->parent);
    reactor_add(r, (event_handler*)delayed_handler_new(1000, quit));
    write(h->out, "main", 5);
    reactor_run(r);
    reactor_destroy(r);
}
```

En este ejemplo disponemos de dos procesos (padre e hijo) que simplemente envían al otro extremo un mensaje de identificación de sí mismos cuando reciben algo previamente. Para iniciar el intercambio de mensajes desde el programa principal enviamos un mensaje al proceso padre.

Nota que inmediatamente después de crear el `process_handler` llamamos a `process_handler_stay_forever_on_child`. Se trata de una función que configura un reactor con un único manejador, el `process_handler` del hijo, y se queda en el bucle de eventos de este reactor hasta el final del programa. Es muy habitual este patrón, por lo que se proporciona una función para implementarlo. El proceso padre no hace nada al invocarla.

Compilar con *reactor*

Por defecto actualmente la biblioteca *reactor* se compila como una biblioteca estática `libreactor.a`. Para poder compilar programas que la usan solo hay que añadir las siguientes banderas al `makefile`:

```
REACTOR=/home/pi/git/src/c/reactor
CFLAGS=-pthread -ggdb -I$(REACTOR)
LDFLAGS=-pthread -L$(REACTOR)/reactor
LDLIBS=-lreactor -lwiringPi -lpthread
```

La variable `REACTOR` tiene la ruta relativa o absoluta donde está instalado reactor. Ajusta su valor si la cambias. El compilador necesita utilizar la bandera `-pthread` para generar correctamente el código de los hilos y la bandera `-I$(REACTOR)` para encontrar los archivos de cabecera. El montador también necesita la bandera `-pthread` para construir correctamente ejecutables con hilos y `-L$(REACTOR)/reactor` para encontrar la biblioteca estática. Finalmente añadimos las bibliotecas que utiliza (*reactor*, *wiringPi* y la biblioteca de manejo de hilos *pthread*).

Todos los ejemplos del taller vienen con un `makefile` completamente funcional pero deliberadamente simple. Analízalos en detalle.

A continuación veremos una serie de ejemplos que pueden ayudarte a entender cómo se trabaja con la biblioteca *Reactor*.

Casos de estudio

Vamos a explicar someramente algunos de los ejemplos incluidos en el software que acompaña al taller. Se trata de ejemplos ligeramente más complejos que los ejemplos triviales, de forma que pueda apreciarse la ventaja de una arquitectura software sólida.

Estos casos de estudio tienen dos propósitos:

- Que el alumno sea consciente de la complejidad real de los sistemas basados en Raspberry Pi. Los ejemplos más evolucionados apenas ocupan unos cientos de líneas de código con módulos de muy bajo acoplamiento y funciones que no superan las 20 líneas.
- Que le sirvan al alumno como fuente de inspiración para que sus proyectos personales sean más creativos y evolucionados.

Antes de nada vamos a ver un mini-caso de estudio en el que realizaremos una de las operaciones más frecuentes cuando se diseña una nueva aplicación, la creación de un nuevo manejador. Esto se va a repetir multitud de veces en nuestros casos de estudio y en la vida real. Es la base de la programación de sistemas: la abstracción. Abstractar es eliminar detalle. Abstractamos para simplificar tanto el uso como la programación de los módulos.

- Cada manejador trata un problema pequeño y nada más. Por tanto su programación es relativamente sencilla y tiene poca relación con el resto del programa.
- Cuando se usa el manejador no es preciso conocer los detalles de cómo se implementa. Solo lo estrictamente necesario.

Sensor analógico

Hemos visto cómo puede medirse una magnitud analógica utilizando un montaje con la pata MISO de la interfaz SPI y una entrada digital. Es hora de convertir esa técnica en un nuevo manejador de eventos para la automatización.

El reto es definir el manejador `analog_handler` para detectar cuando se pasan ciertos umbrales previamente configurados. Debe cumplir los siguientes requisitos:

- Debe poder seleccionarse la pata digital, los umbrales de activación y desactivación, el periodo de muestreo y la frecuencia del reloj.
- Debe medir periódicamente usando el método descrito en el capítulo sobre SPI.
- Debe notificar cuando se supera el umbral de activación y cuando el valor queda por debajo del umbral de desactivación.
- Una vez notificada una activación o una desactivación ya no vuelve a notificar nada hasta que cambie el estado (pasa de activado a desactivado o a la inversa).

Bien, el problema está claro, vamos a resolverlo paso a paso.

Empezando por el final

No me cansaré de repetir esto: *si quieres que funcione, las pruebas primero*. Antes de empezar a hacer nada del nuevo manejador tenemos que hacer un caso de prueba típico. Esto nos ayuda a entender completamente la interfaz de programación que se pretende crear y nos permitirá en el futuro tener cierta confianza en el correcto funcionamiento del manejador.

Es el primer ejemplo de *Reactor*, así que vamos a ayudar un poco. En la carpeta `src/c/ejemplos/analog` tienes un esqueleto de lo que queremos construir. Echa un vistazo a `test_analog_handler.c`. Como ves sigue el convenio de todas las pruebas de la biblioteca *Reactor*, un único archivo que comienza por `test_` junto al nombre del módulo que queremos probar. Procura respetar este convenio, hace que cualquier usuario de tu código encuentre las cosas más fácilmente.

```

#include "analog_handler.h"
#include <reactor/reactor.h>
#include <wiringPi.h>
#include <stdio.h>

static void bajo(analog_handler* this) {
    puts("Bajo limite inferior");
}

static void alto(analog_handler* this) {
    puts("Sobre limite superior");
}

int main(int argc, char* argv[])
{
    wiringPiSetupGpio();
    reactor* r = reactor_new();
    analog_handler* in = analog_handler_new(25, 100, 200,
                                           bajo, alto);

    reactor_add(r, (event_handler*)in);
    reactor_run(r);
    reactor_destroy(r);
    return 0;
}

```

Construimos un `analog_handler` con funciones que simplemente imprimen un mensaje cada vez que detecta el paso de umbral. Solo queda compilarlo con un `makefile`. Como verás en la carpeta `src/c/ejemplos/analog` ya tienes uno:

```

CFLAGS=-pthread -std=c11 -Wall -D_POSIX_C_SOURCE=200809L -I../reactor -ggdb \
-fasynchronous-unwind-tables
LDFLAGS=-L../reactor/reactor -pthread -rdynamic
LDLIBS=-lreactor -lwiringPi -lpthread
CC=gcc

all: test_analog_handler

test_analog_handler: test_analog_handler.o analog_handler.o

clean:
    $(RM) *~ *.o test_analog_handler

```

Esqueleto del manejador

Es ahora y no antes cuando tenemos que plantearnos escribir los archivos `analog_handler.h` y `analog_handler.c`. Se parte de cualquiera de los que ya tienes en *Reactor* y se modifican de acuerdo a las necesidades. Lo primero es decidir qué tipo de `event_handler` es y para ello hay que fijarse en cómo va a notificar los eventos al programa. Como ya sabemos *Reactor* utiliza descriptores de archivo para recibir eventos, por lo que las notificaciones tienen que llegar por este medio. En `analog_handler` no hay descriptores de archivo, solo hay el pin MISO y una salida GPIO. Hay que realizar periódicamente un proceso ciertamente tedioso, en el que hay que contar bits. Todo esto nos da pistas.

Por un lado tenemos que sintetizar eventos en un descriptor de archivo, porque el problema como tal no tiene descriptores. Eso implica usar un `pipe_handler`. Por otro lado hay que realizar un proceso periódico al margen de la tarea principal. Eso nos lleva a un `thread_handler` o un `process_handler`. Dado que no hay que ejecutar nada externo nos decantaremos por un `thread_handler`, que es más eficiente. En *Reactor* tenemos un ejemplo de manejador similar, el de las entradas digitales, solo hay que copiarlo, sustituir `input_handler` por `analog_handler` y quitar todo el código específico.

Empezamos por la cabecera que quedaría así:

```

#ifndef ANALOG_HANDLER_H
#define ANALOG_HANDLER_H
#include <reactor/thread_handler.h>

typedef void (*analog_handler_function)(analog_handler* this);

typedef struct analog_handler_ analog_handler;
struct analog_handler_ {
    thread_handler parent;
    event_handler_function destroy_parent_members;

    // FIXME: Añade todos los atributos que necesites
};

analog_handler* analog_handler_new (int pin, int low, int high,
                                   analog_handler_function low_handler,
                                   analog_handler_function high_handler);
void analog_handler_init (analog_handler* this,
                         int pin, int low, int high,
                         analog_handler_function low_handler,
                         analog_handler_function high_handler);
void analog_handler_destroy (analog_handler* ev);

#endif

```

Esto es lo mínimo: el constructor con y sin reserva de memoria dinámica y el destructor. El nuevo tipo `analog_handler` declara que es una especialización de `thread_handler` porque su primer atributo (*parent*) es un `thread_handler`. Hemos dejado el atributo `destroy_parent_members` porque asumimos que alguna operación de destrucción puede ser necesaria. Ya veremos.

A continuación hacemos lo mismo con `analog_handler.c`:

```

#include "analog_handler.h"

// FIXME: declaraciones de funciones estáticas específicas

analog_handler* analog_handler_new (int pin, int low, int high,
                                   analog_handler_function low_handler,
                                   analog_handler_function high_handler)
{
    analog_handler* h = malloc(sizeof(analog_handler));
    analog_handler_init_members(h, pin, low, high, low_handler, high_handler);
    event_handler* ev = (event_handler*) h;
    ev->destroy_self = (event_handler_function) free;
    thread_handler_start (&h->parent, analog_handler_thread);
    return h;
}

void analog_handler_init (analog_handler* this,
                         int pin, int low, int high,
                         analog_handler_function low_handler,
                         analog_handler_function high_handler)
{
    analog_handler_init_members(this, pin, low, high, low_handler, high_handler);
    thread_handler_start (&this->parent, analog_handler_thread);
}

void analog_handler_destroy (analog_handler* this)
{
    event_handler_destroy((event_handler*)this);
}

// FIXME: Definiciones de funciones estáticas específicas

```

Un `thread_handler` es algo relativamente complejo, porque el orden en que se realizan las operaciones puede afectar al correcto funcionamiento en determinados casos. Procura respetar esta estructura tanto como sea posible para evitarte disgustos.

Los dos constructores (con y sin reserva de memoria) tienen que retrasar la ejecución del hilo hasta el último momento. Por eso extraemos la inicialización de los atributos a otra función diferente (`analog_handler_init_members`).

Hasta este punto lo tienes hecho en la carpeta `src/c/ejemplos/analog` . Sin embargo a partir de aquí te toca trabajar. ¡Manos a la obra!

Funciones específicas

Ya solo queda declarar e implementar las funciones específicas de este manejador. Primero declara las funciones que hemos utilizado en el esqueleto: `analog_handler_thread` y `analog_handler_init_members` . Si tienes dudas consulta `input_handler.c` en la biblioteca *Reactor*. Dado que son funciones que no se exportan al usuario decláralas como `static` .

A continuación haz un esqueleto de ambas funciones justo debajo. Déjalas vacías de momento, solo queremos compilar para detectar problemas lo antes posible. Una vez definidas ejecuta `make` . Verás errores, seguro. Corrige los errores tipográficos, añade las directivas *include* necesarias y consigue que compile sin errores ni advertencias de ningún tipo. Usa `man` si no sabes en qué cabecera está declarada una función del sistema.

Una vez que el programa compila rellena las funciones vacías. ¿Que debe hacer `analog_handler_init_members` ? Lo dice la propia función, inicializar los miembros. Añade a la declaración de la estructura en `analog_handler.h` los miembros que consideres necesarios e inicializa sus valores con los argumentos en `analog_handler_init_members` . No te olvides de configurar adecuadamente el pin MISO y el pin GPIO.

A continuación solo queda el hilo de exploración. El código ya lo tienes en el [capítulo de SPI](#) pero tienes que asegurarte de repetir periódicamente la medida. Usa tantas funciones como sea necesario, no hagas funciones muy largas. Si ves que tienes que usar un `for` y dentro un `if` y dentro otro `if` eso es ya señal de que tienes que dividir en funciones más pequeñas.

Mi sugerencia es copiar de `input_handler.c` tanto como sea posible. Por ejemplo, el hilo puede ser casi igual pero quitando todo lo que tiene que ver con el periodo de muestreo. No hace falta esperar entre muestra y muestra porque el método que vimos en el capítulo de SPI ya necesita esperar a la descarga completa del condensador.

```
static void* analog_handler_thread(thread_handler* h)
{
    analog_handler* in = (analog_handler*) h;
    while(!h->cancel)
        analog_handler_poll(in);
    return NULL;
}
```

Esto es evidentemente incompleto. Hay que definir `analog_handler_poll` . Pero su contenido es directamente lo que vimos en el [capítulo de SPI](#).

Generalizando

Una vez que el programa de prueba esté funcionando llega el momento de hacer algo de autocrítica. ¿Es posible mejorar la interfaz de programación? ¿Es posible con poco trabajo hacer que sea más útil? Por ejemplo:

- Los constructores tienen demasiados argumentos. Es engorroso y propenso a error. Una posible alternativa es pasar una estructura con los datos de configuración.
- El tiempo necesario para estar seguros de que el condensador se ha descargado depende del valor del condensador, habría que ponerlo en un parámetro.
- El reloj SPI a utilizar depende de la magnitud a medir, del tamaño máximo de los buffers pasados a `wiringPiSPIDataRW` y de la precisión necesaria. Habría que dejar también como parámetro de configuración tanto la frecuencia como el tamaño de los buffers.
- Si el condensador no ha tenido tiempo de descargarse es posible que no haya ningún cero en el buffer. Esa condición debe notificarse como un error (elevant una excepción). Lo mismo ocurre si el condensador no ha tenido tiempo de cargarse hasta leer algún uno. Si el último byte leído no es distinto de cero debe notificarse como error (elevant excepción).

Valora tú cuáles de estas críticas deberían arreglarse y propón una solución.

Recapitulando

¿Lo conseguiste? No desistas al primer intento. Solo hay una forma de convertir esto en una rutina, repitiéndolo muchas veces.

De todas formas si te cansas tenemos el ejemplo resuelto. No es buena política leer directamente la solución pero si es tu deseo aquí tienes cómo hacerlo. La solución está en una rama del repositorio que se llama `analog`. Podemos sacarla directamente así:

```
pi@raspberrypi:~/src/c/ejemplos/analog $ git checkout analog -- analog_handler.c
pi@raspberrypi:~/src/c/ejemplos/analog $ git checkout analog -- analog_handler.h
pi@raspberrypi:~/src/c/ejemplos/analog $ █
```

Una vez generalizado el manejador es posible que resulte útil para un uso general. Es el momento de plantearse añadirlo a la propia biblioteca *Reactor*.

Dispositivo MP3

Como primer ejemplo vamos a diseñar un *media player*. Empezaremos por un dispositivo de audio, empleando el manejador `music_player` que ya hemos descrito.

Pero tenemos una *Raspberry Pi* con una GPU Videocore IV capaz de decodificar video FullHD a 30 fps. ¿No es un desperdicio?

Por este motivo haremos una revisión posterior del diseño para mostrar video y música indistintamente. Para ello emplearemos un excelente reproductor específico para Raspberry Pi, el `omxplayer`.

Esto nos dará la oportunidad de implementar otra forma de controlar programas externos, usando su propia entrada/salida estándar.

Construir un *audio player*

Tienes un ejemplo de uso del manejador `media_player` en `test/test_media_player.c`. La única diferencia con un media player convencional es que un dispositivo portátil no puede tener conectado un teclado. Partiendo de este ejemplo y de `reactor/test/test_input.c` modifica `ejemplos/mp3/audio_player.c` para que se controle con botones y no con el teclado. Veamos un posible esqueleto.

```
#include <reactor/reactor.h>
#include <reactor/music_player.h>
#include <reactor/input_handler.h>

enum {
    NEXT= 18,
    PREV = 23,
    PLAY = 24,
    PAUSE = 25
};

int main(int argc, char* argv[])
{
    char* path = (argc > 1
                  ? argv[1]
                  : "/usr/share/scratch/Media/Sounds/Music Loops");
    music_player* mp = music_player_new(path);

    void press(input_handler* ev, int key) {
        // Actuar sobre mp según la tecla
        // ...
    }
    void release(input_handler* ev, int key) {}
    int keys[] = { NEXT, PREV, PLAY, PAUSE };
    wiringPiSetupGpio();
    input_handler* ih = input_handler_new(keys, sizeof(keys),
                                          press, release);

    reactor* r = reactor_new();
    reactor_add(r, (event_handler*)mp);
    reactor_add(r, event_handler_new(0, keyboard));
    reactor_run(r);
    reactor_destroy(r);
    console_restore(0, state);
    return 0;
}
```

Al igual que en `test_music_player.c` usamos las funciones anidadas de las versiones modernas de C para pasar el *music_player* a las funciones de *press* y *release*. El mismo efecto (pero con más código) puede conseguirse derivando de *input_handler* y añadiendo el *music_player* como un atributo más.

Este tipo de aplicaciones son ideales para la Raspberry Pi Zero pero es un desperdicio para una Raspberry Pi 3. Por lo menos vamos a aprovechar que tiene un mayor factor de forma para ponerle un reproductor de video.

Construir un *video player*

Para construir un reproductor de video necesitaríamos añadirle una pantalla a la Raspberry Pi. Las opciones disponibles son:

- Pantalla HDMI. Puede ser la opción más barata y no consume pines adicionales. Por ejemplo, en BangGood una de 5 pulgadas con *touch panel* no llega a 40€.
- [La pantalla oficial de la Raspberry Pi Foundation](#) que utiliza la interfaz DSI de la Raspberry Pi y tiene panel táctil multipunto. Puede usarse simultáneamente con un monitor HDMI. Aunque el precio oficial es de 60\$ es difícil encontrarla a ese precio. RS la tiene actualmente por 67.19€
- [PiTFT Plus de Adafruit](#) que utiliza la interfaz SPI y dos o tres pines de GPIO. Es una pantalla que tiene el mismo factor de forma que los modelos B, lo que desde el punto de vista estético es muy conveniente, pero el consumo excesivo de pines reduce las posibles aplicaciones. El coste depende de la resolución, entre 35\$ y 45\$. Para los modelos anteriores a la 2B tal vez te interese [este clon chino de BangGood](#) a tan solo 10.81€.

Para construir un video player hemos hecho un nuevo manejador a partir de `process_handler`, pero redirigiendo entrada y salida con dos pipes, como hace el `pipe_handler`. Esto es así porque controlamos el programa *omxplayer*, incluido en la propia Raspberry Pi empleando su propia entrada estándar. Es decir, simulamos pulsaciones de teclas. El programa *omxplayer* es un reproductor multimedia que aprovecha todas las características del Videocore IV, incluyendo los codecs privativos que se venden en la tienda online de la Raspberry Pi Foundation. Esto permitiría reproducir video full-HD a 30 fps.

Algunos programas no permiten la redirección de la entrada a una *pipe*. Es típico en los casos en los que se utilizan capacidades avanzadas del terminal, como cuadros de diálogo, animaciones de texto, ventanas, etc. En esos casos lo más probable es que requiera una entrada y salida estándar con capacidades de terminal. Es el momento de usar *pseudoterminales*. Ampliaremos esto más adelante, en el sitio web del taller. Sigue las novedades.

En `ejemplos/mp3/test_video_player.c` tienes un ejemplo de cómo usar el nuevo manejador que hemos hecho para este caso. Tu trabajo ahora es convertirlo en un video player con botones. Combina ahora ese manejador `video_player` con el `input_handler` de forma similar a como hiciste con el *audio player*.

Piano de juguete

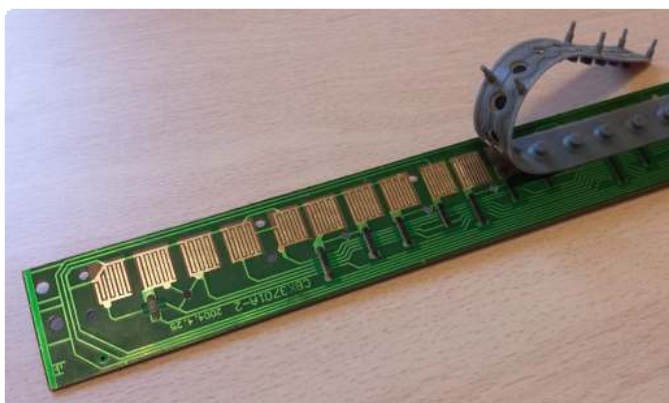
Vamos a analizar como caso de estudio un piano de juguete. Supongamos que tenemos un muchachito travieso que se ha subido en su piano de juguete y ha estado saltando encima de él hasta que ha dejado de sonar. Estas cosas pasan, os lo aseguro, ya no hacen los pianos como antes. Supongamos que el muchachito está muy arrepentido y te pide con su mejor cara de angelito que *le pongas pilas al piano*. No vale la pena explicarle que el problema es mayor, hay que hacer algo. Armados con un destornillador y un poco de paciencia conseguimos desmontarlo entero e identificamos el teclado de membrana que hay bajo las teclas del piano.



Piano infantil como el que vamos a modificar.

Las teclas están físicamente dispuestas como una línea pero lógicamente dispuestas de forma matricial, como muestra la figura adjunta. Hay ocho columnas y tres filas. Cuando se pulsa una de las teclas se cortocircuita la columna correspondiente con la fila correspondiente. Es así de simple.

Lo que vamos a hacer es eliminar el controlador actual del piano y sustituirlo por una Raspberry Pi. Lo ideal sería poner una Raspberry Pi Zero, pero eso es un detalle menor, puesto que el diseño y el software es exactamente igual con cualquier otro modelo actual. Para los altavoces podemos usar cualquier mecanismo utilizado en un ordenador convencional. A mi me gusta especialmente la idea de un altavoz Bluetooth, que puede ser externo si queremos. Se puede conseguir uno resistente al agua por poco más de seis euros en Banggood.com.



Teclado de membrana bajo las teclas del piano.

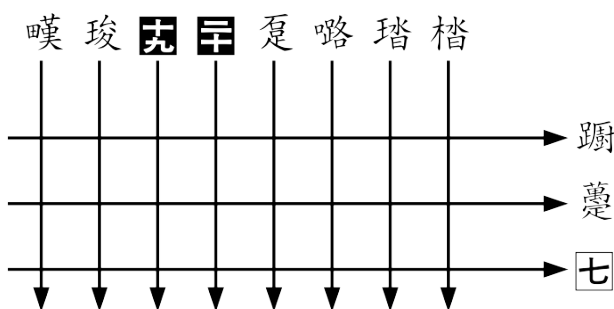
Bueno, ya tenemos el primer concepto, solo nos falta el software para tener el prototipo. Hay dos cosas que debemos hacer:

- Detectar las pulsaciones de tecla evitando los rebotes típicos de los teclados de membrana.
- Tocar notas correspondientes a cada tecla desde el momento que se aprieta hasta el momento que se libera.

La biblioteca *Reactor* tal y como la hemos explicado hasta ahora implementa un tipo de entradas

`input_handler` que vale perfectamente para botones

independientes. Sin embargo con la disposición matricial no es inmediato que funcione sin modificar nada. Vamos a explicar cómo enfrentaríamos este problema suponiendo que la biblioteca no lo contempla. Está claro que tenemos que hacer cambios en `input_handler`, pero ¿cómo?



Disposición matricial del teclado de membrana.

El otro problema, tocar notas, no es tan simple como parece a primera vista. No es cuestión de tocar un *mp3* correspondiente a cada nota y listos. Podemos apretar varias notas a la vez y deberían sonar todas simultáneamente. Eso exige un proceso de síntesis de sonidos y mezclado digital.

Todo se puede hacer, pero en este taller seguimos la filosofía KISS (*Keep It Simple, Stupid*). No vamos a trabajar en algo que ya está resuelto, bastante trabajo tenemos con lo no resuelto. Raspbian tiene preinstalado *Sonic-Pi*, una fantástica herramienta docente del propio Laboratorio de Computadores de la Universidad de Cambridge, el mismo que hizo Raspberry Pi. Nos basta un simple [tutorial](#) para ver que hace todo lo que necesitamos y muchísimo más. Mejor, en el futuro le añadiremos funcionalidad.

Solo falta ver cómo podemos usar *Sonic-Pi* desde nuestro programa. Examinando en detalle la documentación descubrimos que Sonic-Pi utiliza otro programa para generar el sonido, [SuperCollider audio synthesis server](#), y se comunica con él a través de mensajes UDP hacia el puerto 4556. Fantástica idea, vamos a usar el `connector` de la biblioteca *Reactor* para esto.

SuperCollider tiene un ejecutable independiente llamado `scsynth` con soporte de conexiones remotas UDP o TCP e incluso un cliente de línea de órdenes llamado `sclang` (ver página de manual).

Todos los elementos están claros, solo falta sentarse a diseñar el pegamento:

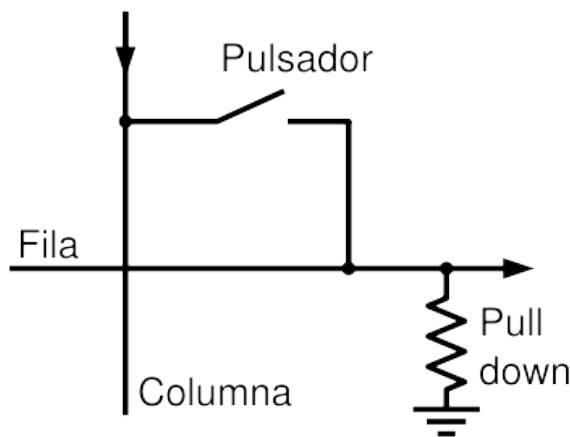
- El servidor `scsynth` debe arrancarse previamente. Podemos echar una ojeada al código de Sonic-Pi para ver qué opciones serían razonables. O más simple aún, podemos capturar los mensajes usando *wireshark*.
- Nuestro programa debe explorar el teclado para detectar pulsaciones y levantamientos de tecla.
- Cuando se presiona una tecla debe comunicarse con `scsynth` para añadir a lo que ya se está tocando una nota más.
- Cuando se libera una tecla debe comunicarse con `scsynth` para quitar de lo que se esté tocando la nota correspondiente a esta tecla.

Exploración del teclado

La estrategia para leer una matriz como la del teclado es conectar las columnas con salidas digitales y las filas con entradas digitales (o al revés). En lo sucesivo asumiremos que se conecta como muestra la figura. Fíjate en que la entrada digital correspondiente a cada fila tiene configurado un *pull-down* para que en caso de que no haya tecla pulsada se lea como un cero lógico.

El barrido se realiza activando una sola columna cada vez y comprobando si hay alguna fila a uno lógico. Si se encuentra una entonces es que se ha pulsado la tecla correspondiente al cruce de la columna activa y la fila leída como uno.

El proceso es muy similar al `input_handler` pero al bucle de consulta de las entradas hay que añadir el bucle de activación de cada columna y la notificación debe incluir tanto la fila detectada como la columna activa. Por tanto partiremos del código de `input_handler` y lo modificaremos para acomodarlo a nuestras necesidades.



Esquema eléctrico de cada tecla.

Todo el código de este ejemplo lo tienes disponible en la carpeta `src/c/ejemplos/piano`. El código del `input_handler` modificado está en `matrix_handler.h` y `matrix_handler.c`. A continuación resumimos los cambios que hemos realizado:

- Hemos renombrado los archivos y todos los `input_handler` por `matrix_handler`.
- El atributo `inputs` pasa a ser `rows` y se crea un nuevo atributo `cols`. El contador `ninputs` se desglosa en `nrows` y `ncols`.
- El atributo `values` pasa a ser un array de `nrows*ncols` valores.
- En lugar de notificar el pin activo notificamos el número de tecla.
- Exploramos todas las filas para cada columna activa en solitario.

Eso es todo pero no debemos parar ahí. Acabamos de hacer un nuevo handler que solapa extraordinariamente con el antiguo `input_handler`. Nunca debemos dejar código replicado. En este caso podemos convertir el `input_handler` en un caso particular del `matrix_handler` con cero columnas. Para ello tenemos que considerar este caso especial en la función de exploración `matrix_handler_poll`.

Info No tengas miedo de cambiar *Reactor* para adaptarla a tus necesidades. Es mucho más importante que no haya código replicado que preservar el código de la biblioteca intacto. Si haces manejadores que pueden ser útiles a más gente considera compartirlos como hemos hecho nosotros. Contacta con los autores para incorporarlos a la distribución oficial.

Hasta la prueba del teclado es casi idéntica a la de `input_handler`.

```
#include <reactor/reactor.h>
#include "matrix_handler.h"
#include <wiringPi.h>
#include <stdio.h>

static void press(matrix_handler* ev, int key) {
    printf("Press %d\n", key);
}

static void release(matrix_handler* ev, int key) {
    printf("Release %d\n", key);
}

int main() {
    int rows[] = { 4, 17, 27, 22 };
    int cols[] = { 18, 23, 24, 25 };
    wiringPiSetupGpio();
    reactor* r = reactor_new();
    reactor_add(r, (event_handler*) matrix_handler_new(rows, sizeof(rows),
                                                         cols, sizeof(cols),
                                                         press, release));

    reactor_run(r);
    return 0;
}
```

Control remoto de SuperCollider

Es el momento de aprender algo sobre *SuperCollider* y de su control remoto en doc.sccode.org. El control remoto utiliza una versión simplificada del protocolo [Open Sound Control](#).

- Todos los datos se codifican en *big-endian* (*network byte order*).
- Si es TCP cada mensaje va precedido por un entero de 32 bits que contiene la longitud del mensaje, en UDP no hace falta. A continuación aparece una orden (*command*) o un paquete de órdenes (*bundle*).
- Cada orden (*command*) consiste en una cadena que representa la orden, seguida de una coma y una lista de letras que representan los tipos de los argumentos. A continuación vendrían los valores de los argumentos en *big-endian*. Las cadenas codifican el `\0` terminador y además requieren relleno (*padding*) hasta el siguiente múltiplo de cuatro bytes.
- Cada paquete de órdenes (*bundle*) empieza con la cadena `#bundle` (incluido el `'\0'` terminador de las cadenas C). A continuación una marca temporal de 64 bits en el formato usado por NTP (*Network Time Protocol*) y a continuación todas las órdenes incluidas en el paquete codificadas igual que en un mensaje normal (con la longitud precediendo a cada orden). Los paquetes de órdenes no van a ser necesarios.

La lista completa de las órdenes soportadas está disponible en doc.sccode.org. Algunas órdenes reciben respuestas usando el mismo protocolo.

Básicamente el procedimiento puede extraerse de [esta página](#). En el momento en que se pulsa una tecla enviaremos una orden `/s_new`. Previamente hay que configurar los `SynthDef` ya compilados en una carpeta accesible por `scsynth`.

Antes de nada vamos a hacer pruebas para asegurarnos del funcionamiento de los mensajes. Arranca `wireshark` y empieza una captura de paquetes en la interfaz de *loopback* `lo`. Después arranca `sonic-pi` y escribe el siguiente programa:

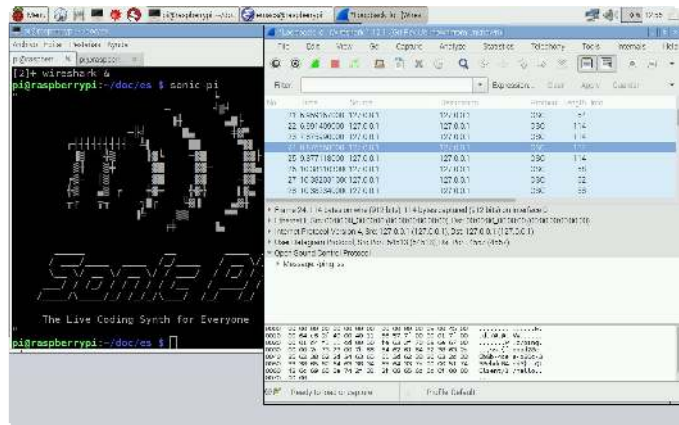
```
use_synth :piano
play :e1
```

Y pulsamos el botón *Run*. Ya podemos analizar la captura. La mayoría del tráfico capturado indica que se trata de protocolo *OSC (Open Sound Control)*, pero si exploramos un poco veremos que hay varios destinatarios. ¿Cómo sabemos cuáles son los mensajes que corresponden a `scsynth` ? Basta mirar cómo se ha ejecutado en un terminal:

```
pi@raspberrypi:~/src/c $ ps -ef | grep scsynth
pi      2717  2687  7 13:09 pts/1    00:00:01 scsynth -u 4556 -a 64 -m 131072
-D 0 -R 0 -l 1 -z 128 -c 128 -U /usr/lib/SuperCollider/plugins:/opt/sonic-pi/app
/server/native/raspberry/extra-ugens/ -i 2 -o 2
pi      2753  2746  0 13:10 pts/2    00:00:00 grep --color=auto scsynth
```

Como puedes observar se ejecuta con la opción `-u 4556` que indica el puerto UDP donde escucha. Por tanto el tráfico que nos interesa es el que tiene como destino u origen el puerto UDP 4556. Volvamos a *Wireshark*, en el recuadro *Filter* basta teclear la expresión `udp.port == 4556` y pulsar sobre el botón *Apply*. A continuación se muestra todo el tráfico destinado al puerto 4556 de forma esquemática. Guarda la captura entera para consultarla cuando sea necesario.

```
[ "/clearSched" ]
[ "/g_freeAll", 0 ]
[ "/notify", 1 ]
[ "/d_loadDir", "/opt/sonic-pi/etc/synthdefs/compiled" ]
[ "/sync", 1 ]
[ "/b_allocRead", 0, "/opt/sonic-pi/etc/buffers/rand-stream.wav", 0, 0 ]
[ "/clearSched" ]
[ "/sync", 1 ]
[ "/g_freeAll", 0 ]
[ "/g_new", 2, 0, 0 ]
[ "/g_new", 3, 2, 2 ]
[ "/g_new", 4, 2, 3 ]
[ "/g_new", 5, 3, 2 ]
[ "/s_new", "sonic-pi-mixer", 6, 0, 2, "in_bus", 10 ]
[ "/sync", 1 ]
[ "/g_new", 7, 1, 4 ]
[ "/s_new", "sonic-pi-basic_mixer", 8, 0, 2, "amp", 1, "amp_slide", 0.1, "amp_slide_shape", 1, "amp_slide_curve", 0, "in_bus", 12, "amp", 0.3, "out_bus", 10 ]
[ "#bundle"
  [ "/s_new", "sonic-pi-piano", 9, 0, 7, "note", 28.0, "out_bus", 12 ]]
[ "/n_set", 8, "amp_slide", 1.0 ]
[ "/n_set", 8, "amp", 0 ]
[ "/n_free", 8 ]
[ "/n_free", 7 ]
[ "/quit" ]
```



Captura de *Wireshark* mostrando todo el tráfico OSC de *Sonic-Pi*.

Hay mucho más de lo que uno podría pensar a primera vista. No solo se instancia el piano sino también un `sonic-pi-mixer` y un `sonic-pi-basic_mixer`. La nota no se envía en una orden OSC sino en un paquete en el que solo está esa nota, parece un poco redundante. A priori se observan otras redundancias, como la repetición de `clearSched` y `g_freeAll`, o la activación de las notificaciones (`notify`) y varios mensajes de petición de estado (`status`) que no hemos puesto para hacerlo más legible. Los mensajes `sync` hacen pensar que es necesario esperar a que lo anterior se complete para evitar problemas en los siguientes mensajes. En general la aproximación que recomendamos es implementar todo lo que hace *Sonic-Pi* y cuando todo funcione entonces vamos simplificando.

Ya tenemos claro los elementos. Hay que ejecutar `scsynth` y usar un `udp_connector` para enviar mensajes. Cuando nuestro programa termine `scsynth` también debe hacerlo. Una forma sencilla de conseguirlo es con un `process_handler`, aunque no necesitemos reaccionar ante lo que imprima el programa.

Osea para este ejemplo necesitamos un `process_handler` y un `udp_connector`, ¿cómo lo agrupamos? ¿Definimos otro manejador de *Reactor*? La respuesta está en cómo queremos lidiar con la entrada. *Reactor* es un patrón para definir comportamientos reactivos, si no hay que reaccionar no es necesario usarlo. Los manejadores no obstante podemos usarlos por mera conveniencia.

¿Y si llegan datos del otro extremo? En el caso del proceso de `scsynth` no hay nada que podamos hacer con la salida estándar, así que redirigiremos toda la salida a `/dev/null`. Sin embargo el control remoto de `scsynth` puede generar respuestas y notificaciones (ver la [guía de referencia de órdenes](#)). Por ejemplo, el mensaje `/d_loadDir` es asíncrono y no termina hasta que se recibe la respuesta `/done`.

Por tanto ¿qué es nuestro sintetizador? Se trata de un `udp_connector` especializado. Un `udp_connector` que automáticamente arranca otro proceso aprovechando un `process_handler` para ello. Un conector que al destruirse envía el mensaje `/quit` a `scsynth` luego destruye el `process_handler` correspondiente. Un `udp_connector` con métodos especializados para enviar y recibir mensajes OSC. Ahora ya podemos sentarnos a escribir código.

Info Es conveniente recapacitar un poco sobre el proceso de diseño que hemos seguido. Nuestro problema era la síntesis de sonido y ha terminado siendo las comunicaciones con un servidor. Esto es muy frecuente en la vida real, lo que vemos como problema no siempre es el problema si aplicamos ciertas dosis de imaginación y pensamiento lateral.

La mayor parte del trabajo tiene que ver con la codificación de mensajes OSC iguales que los que tenemos en la captura de *Wireshark*. En general optaría por utilizar una implementación de OSC ya disponible, como *liblo*. Esto es una inversión a largo plazo porque permite integrar nuestro piano con otras aplicaciones OSC. Lo que queremos es ser capaces de ejecutar algo como esto:

```
synth_handler_send(synth, "/d_loadDir",
                  "/opt/sonic-pi/etc/synthdefs/compiled");
```

Dado que los mensajes OSC tienen número variable de argumentos nuestra función `synth_handler_send` también tiene que tener número variable de argumentos, como `printf`. Echa un vistazo a la página de manual de `stdarg` y al archivo `synth_handler.c` para ver cómo se implementa. El caso es que los argumentos acabarán en una lista de tipo `va_list` que es lo que vamos a usar para componer el mensaje. Para facilitar el uso de *liblo* añadiremos algún parámetro para indicar tipos y marcar el final de argumentos. Se trata de medidas para detectar posibles errores de programación. Nuestro ejemplo quedaría así:

```
synth_handler_send(synth, "/d_loadDir", "s",
                  "/opt/sonic-pi/etc/synthdefs/compiled",
                  LO_ARGS_END);
```

En los archivos `synth_handler.h` y `synth_handler.c` se incluye toda la implementación de este manejador. La interfaz de programación es extremadamente simple:

```
synth_handler* synth_handler_new(synth_handler_function handler);
void synth_handler_init(synth_handler* this,
                      synth_handler_function handler);
void synth_handler_destroy(synth_handler* this);
void synth_handler_send(synth_handler* h, const char* cmd,
                      const char* types, ...);
void synth_handler_sync(synth_handler* this);
```

La función `synth_handler_sync` espera hasta que se han recibido respuesta de todas las operaciones asíncronas pendientes. Para ello simplemente utiliza un mensaje `/sync`.

Juntando todo

Con los dos nuevos manejadores terminar la aplicación es francamente trivial. El código de este ejemplo está en `piano.c`. Solo mostramos aquí la parte más significativa:


```
static void press(matrix_handler* ev, int key)
{
    static int n = 100;
    synth_handler_send(synth,
                       "/s_new", "siiisfsi",
                       "sonic-pi-piano", n++, 0, 7,
                       "note", 48.0 + key,
                       "out_bus", 12,
                       LO_ARGS_END);
}

int main()
{
    int rows[] = { 4, 17, 27, 22 };
    int cols[] = { 18, 23, 24, 25 };

    wiringPiSetupGpio();
    synth = synth_handler_new(do_nothing);
    reactor* r = reactor_new();
    reactor_add(r, (event_handler*) matrix_handler_new(rows, NELEMS(rows),
                                                         cols, NELEMS(cols),
                                                         press, release));

    reactor_add(r, (event_handler*) synth);
    init_synth(synth);
    reactor_run(r);
    reactor_destroy(r);
    return 0;
}
```

Epílogo

Hemos implementado un piano básico, pero las capacidades disponibles son ahora muy superiores a las del piano de juguete. El sonido es de mejor calidad, con efecto *aftertouch*, y podemos añadir todos los ritmos de fondo que queramos. Podemos cambiar el instrumento por cualquiera de la extensa biblioteca disponible para Sonic-Pi. Podemos sincronizar varios pianos de juguete para que toquen de forma coordinada con una única salida de sonido. Podemos implementar juegos del estilo del *Guitar Hero*. El límite lo marcas tú.

Desarrollo en Python

Python es un excelente lenguaje de programación que goza de una merecida fama de resultar fácil de aprender. Esto no significa que se trate de un lenguaje de juguete, todo lo contrario. Se utiliza extensivamente en la infraestructura de Google o en Dropbox, como sustituto de Matlab en multitud de aplicaciones de cálculo científico, o incluso para cálculos simbólicos. Tiene una muy amplia colección de bibliotecas para casi cualquier cosa que te propongas.

Es la primera vez que impartimos el taller en Python así que estamos seguros de que tendrá muchas esquinas que mejorar. Se aceptan sugerencias, por supuesto.

A diferencia de los capítulos de C no vamos a desplegar toda la gama de posibilidades de programación que se ofrecen, sino que nos limitaremos a la opción que consideramos *mas pitónica* o con mas futuro.

En general la Raspberry Pi Foundation parece que apoya firmemente la biblioteca *GPIO Zero* como la forma más intuitiva y directa de programar aplicaciones con una Raspberry Pi. Sin embargo no es suficiente porque en la actualidad no implementa ni mucho menos todas las capacidades necesarias. Eso nos lleva a describir otras alternativas solo en aquellos aspectos que no son cubiertos por *GPIO Zero*.

Entorno de desarrollo

En general te recomendamos que utilices el entorno de desarrollo *IDLE*, que viene incluido en Python. Por comodidad tienes un botón en la barra de tareas. Ten en cuenta que el botón corresponde a *IDLE* de Python 3. Si prefieres usar la versión 2 utiliza la barra de menús.

Programando los periféricos

Vamos a ver ahora una serie de ejemplos de programación en Python de los elementos que hemos visto en el capítulo 2. Nuestro objetivo es ofrecer una referencia de lo que hoy en día se utiliza para programar la Raspberry Pi usando lenguaje Python. Eso implica que seleccionaremos en cada caso la que consideramos la mejor alternativa. No dudes en enviarnos cualquier sugerencia si no estás de acuerdo con nuestra elección.

Cuando te enfrentas a un nuevo proyecto tienes que entender bien todos los elementos involucrados. Esto normalmente implica explorar, probar y leer. Tu primer objetivo tiene que ser reducir la incertidumbre, tener los detalles suficientes para no tener que mirar la hoja de datos de los dispositivos a cada paso cuando estemos diseñando la aplicación.

No tomes estos ejemplos como una muestra de cómo se debe programar, eso lo veremos más adelante. Se trata de que aprendas a usar los periféricos desde tu lenguaje de programación favorito. Por tanto intentamos que el código sea lo más simple y directo posible, no el más mantenible, ni siquiera el más legible. Primamos sobre todo que se vea el uso de la API ofrecida por cada biblioteca.

Entradas y salidas digitales en Python

Para programar entradas y salidas digitales en Python tenemos también una amplia variedad de bibliotecas:

- La biblioteca *GPIO Zero* proporciona una interfaz abstracta muy sencilla de usar para una amplia variedad de periféricos. Es especialmente cómoda para el uso de entradas y salidas digitales.
- La biblioteca *RPi.GPIO* es un módulo que proporciona acceso exclusivamente a las entradas y salidas digitales. Actualmente no soporta SPI, I2C, hardware PWM o comunicaciones serie, aunque está previsto que lo soporte en el futuro.
- La biblioteca *wiringPi2* no es más que un envoltorio de la biblioteca C del mismo nombre. Por tanto gran parte de lo que decimos en el capítulo anterior sobre *wiringPi* es también aplicable a Python.
- La biblioteca *pigpio* es un envoltorio de la interfaz a *pigpiod* en C. El programa *pigpiod* ya hemos tenido ocasión de utilizarlo en el segundo capítulo, a través de su interfaz de línea de órdenes `pigs`. El módulo Python ofrece unas características similares y permite una conexión remota a *pigpiod*. Eso significa que podemos ejecutar nuestro programa en un ordenador convencional y que éste se comunique con la Raspberry Pi para realizar la interacción con el mundo físico.

En este capítulo describiremos exclusivamente la primera, aunque animamos a experimentar con otras alternativas una vez que se conozcan los fundamentos.

GPIO Zero fue desarrollada fundamentalmente por Ben Nuttall de la Raspberry Pi Foundation y Dave Jones, el creador de la biblioteca *picamera*. Sigue fielmente el propósito de la Fundación, impulsar la docencia de la electrónica y la informática en edades tempranas. Por tanto es extremadamente fácil de usar.

Luces y botones

Un aspecto importante de *GPIO Zero* es que cambia el foco del programador hacia el sistema. No se piensa en términos de los periféricos del BCM2835, sino en términos de los componentes del sistema. Esto hace que su uso sea tremendamente intuitivo.

```
from gpiozero import LED, Button
from signal import pause

led = LED(17)
button = Button(3)

button.when_pressed = led.on
button.when_released = led.off

pause()
```

Prácticamente se explica solo. Utiliza una interfaz declarativa, en la que simplemente se describe lo que tiene que pasar cuando suceden eventos en los componentes del sistema. No se indica explícitamente qué pines son salidas o entradas, pero lo hace de forma automática al indicar que en GPIO17 hemos puesto un LED y en GPIO3 un pulsador. No se indica explícitamente que se active un *pull-up*, pero lo hace por el mero hecho de decir que en la pata GPIO3 hemos puesto un pulsador. Podemos cambiar el *pull-up* por un *pull-down* añadiendo un parámetro en el constructor `button = Button(3, pull_up = False)`.

La línea `button.when_pressed = led.on` indica que cuando ocurra el evento *pressed* sobre el botón se invoque a la función `led.on()`. Es así de simple. Evidentemente por debajo hay un hilo que está consultando el estado de la entrada digital, pero se oculta completamente al usuario.

Es muy frecuente que una vez declarado todo lo que debe ocurrir ya no tengamos nada más que hacer. Si no hacemos nada más el programa terminaría. Así que no nos queda más remedio que quedarnos en algún tipo de bucle que no haga nada. La forma más conveniente en Python es llamar a la función `signal.pause()` que simplemente espera hasta que reciba una señal. Ya hemos hablado de señales cuando describíamos la orden `kill`. Un simple *Control-C* (interrupción de usuario) es recibido como una señal y por tanto saldría.

Cada vez que asociamos un dispositivo a un *pin* la biblioteca lo registra como ocupado y generaría una excepción si intentamos asociar dos dispositivos al mismo pin. Esto es extremadamente útil para evitar errores de programación, pero en ocasiones tenemos multiplexados varios dispositivos a una misma pata. Para eso se proporciona el método *close*, que libera el pin para otros usos:

```
bz = Buzzer(16)
bz.on()
bz.off()
bz.close()
led = LED(16)
led.blink()
```

La lista de las clases y dispositivos soportados crece continuamente, así que la mejor forma de estudiar la biblioteca es emplear los métodos de consulta de documentación de Python.

Explorando GPIO Zero

No hay mejor forma de explorar la jerarquía de clases de GPIO Zero que leyendo el código con ayuda de IDLE. Hay multitud de ejemplos en los comentarios y documentación de cada clase de la biblioteca. Simplemente recuerda que las funciones que empiezan por un carácter de subrayado (`_`) no están pensadas para ser usadas fuera de la clase.

Abre IDLE y en el menú *File* selecciona *Open Module*. Escribe `gpiozero`. Aparecerá el archivo principal de la biblioteca, que importa todos los demás. Selecciona otra vez el menú *File* pero ahora pincha en *Open....* Elige `input_devices.py`. En la nueva ventana vuelve a seleccionar el menú *File* pero ahora pincha en *Class Browser*. Aparecerá una ventana con todas las clases definidas en este archivo. El mismo procedimiento lo puedes repetir con `output_devices.py`, `spi_devices.py` o cualquier otro archivo de la biblioteca.

Es importante conocer cómo está hecha la biblioteca porque es muy probable que nos enfrentemos a algún dispositivo que no está actualmente modelado en GPIO Zero. Veamos la estructura de `Button` y `LED`. Navega primero por las clases de `output_devices.py`

```
class LED(DigitalOutputDevice):
    pass
```

La clase `LED` no es más que un sinónimo de `DigitalOutputDevice`.

```
class DigitalOutputDevice(OutputDevice):
    def __init__(self, pin=None, active_high=True, initial_value=False):
        # ...

    @property
    def value(self):
        return self._read()

    @value.setter
    def value(self, value):
        self._stop_blink()
        self._write(value)

    def close(self):
        self._stop_blink()
        super(DigitalOutputDevice, self).close()

    def on(self):
        self._stop_blink()
        self._write(True)

    def off(self):
        self._stop_blink()
        self._write(False)

    def blink(self, on_time=1, off_time=1, n=None, background=True):
        # ...
```

Tiene métodos `on`, `off` y `blink` (parpadear), así como una propiedad `value` que cuando se escribe cancela el parpadeo si lo hay y escribe el valor. Las propiedades funcionan igual que los atributos pero al asignar valores o leer valores se invocan funciones en lugar de acceder a los valores directamente.

Al derivar de `OutputDevice` incorpora toda la funcionalidad de esta clase:

```
class OutputDevice(SourceMixin, GPIODevice):
    def __init__(self, pin=None, active_high=True, initial_value=False):
        # ...

    def _write(self, value):
        if not self.active_high:
            value = not value
        try:
            self.pin.state = bool(value)
        except AttributeError:
            self._check_open()
            raise

    def on(self):
        self._write(True)

    def off(self):
        self._write(False)

    def toggle(self):
        with self._lock:
            if self.is_active:
                self.off()
            else:
                self.on()

    @property
    def value(self):
        return super(OutputDevice, self).value

    @value.setter
    def value(self, value):
        self._write(value)

    @property
    def active_high(self):
        return self._active_state

    @active_high.setter
    def active_high(self, value):
        self._active_state = True if value else False
        self._inactive_state = False if value else True
```

Básicamente proporciona los mismos servicios que `DigitalOutputDevice` salvo por el parpadeo. Además implementa la lógica necesaria para tratar las señales desde un punto de vista lógico, independientemente de si son activas a nivel alto o bajo.

Pero interesa especialmente las clases de las que deriva, `SourceMixin` y `GPIODevice`. La primera puede encontrarse en `mixins.py`. Un *mixín* es una clase que incorpora funcionalidad adicional a la clase a la que se la aplique, que depende de la propia clase. En este caso añade una propiedad `source` que dado un iterable lo recorre y asigna el valor leído a la propiedad `value`. Cada lectura se separa un tiempo que es definible en otra propiedad `source_delay`. Veremos en seguida cómo coopera con los dispositivos de entrada.

La otra clase ancestro es `GPIODevice` que está definida en `devices.py`:

```
class GPIODevice(Device):
    """
    Extends :class:`Device`. Represents a generic GPIO device and provides
    the services common to all single-pin GPIO devices (like ensuring two
    GPIO devices do no share a :attr:`pin`).

    :param int pin:
        The GPIO pin (in BCM numbering) that the device is connected to. If
        this is ``None``, :exc:`GPIOPinMissing` will be raised. If the pin is
        already in use by another device, :exc:`GPIOPinInUse` will be raised.
    """
    ...
```

Es decir, simplemente se encarga de garantizar que el pin no está siendo usado por otro dispositivo. El resto lo delega en `Device` que está en el mismo archivo:

```
class Device(ValuesMixin, GPIOBase):
    @property
    def value(self):
        raise NotImplementedError

    @property
    def is_active(self):
        return bool(self.value)
```

Define una propiedad `value` que no está implementada (se implementa en las clases derivadas) y deriva de dos nuevas clases (`ValuesMixin` y `GPIOBase`).

`ValuesMixin` está en `mixins.py` . Simplemente añade la propiedad `values` que se construye como un generador que lee una y otra vez la propiedad `value` . Esto permite conectarlo con la propiedad `source` . Piensa por ejemplo qué hace el siguiente fragmento:

```
led1 = LED(17)
led2 = LED(18)
led1.source = led2.values
```

El resto de clases no interesa de momento. La biblioteca llega a una sofisticación notable impidiendo que un usuario ponga atributos sin darse cuenta, por un error sintáctico. Por ejemplo, prueba esto:

```
from gpiozero import *
led1 = LED(17)
led2 = LED(18)
led1.surce = led2.values
```

En Python esta sintaxis sería completamente legal, el error tipográfico se traduciría en que el programa no funciona y el `led1` tiene un atributo extraño `surce` . En GPIO Zero es un error intentar añadir atributos después de la construcción.

Vamos a explorar un poco la clase `Button` . Navega ahora por las clases de `input_devices.py` .

```
class Button(HoldMixin, DigitalInputDevice):
    def __init__(
        self, pin=None, pull_up=True, bounce_time=None,
        hold_time=1, hold_repeat=False):
        # ...
```

Osea, que simplemente es una combinación de `HoldMixin` y `DigitalInputDevice` . La primera es un *mixín* y por tanto está en `mixins.py` .

```
class HoldMixin(EventsMixin):
    """
    Extends :class:`EventsMixin` to add the :attr:`when_held` event and the
    machinery to fire that event repeatedly (when :attr:`hold_repeat` is
    ``True``) at intervals defined by :attr:`hold_time`.
    """
    ...
```

Simplemente añade el evento `when_held` a la clase a la que se aplica. Si se mantiene pulsado el botón por un tiempo configurable en la propiedad `hold_time` entonces se invocará a la función `when_held`.

La clase `EventsMixin` es otro *mixín* que añade los eventos `when_activated` y `when_deactivated`, así como los métodos `wait_for_active` y `wait_for_inactive`.

Por otro lado `DigitalInputDevice` es una combinación de `EventsMixin` (que ya conocemos) y `InputDevice`:

```
class DigitalInputDevice(EventsMixin, InputDevice):
    def __init__(self, pin=None, pull_up=False, bounce_time=None):
        ...
```

En cuanto a `InputDevice` lo único que añade a un `GPIODevice` es la propiedad `pull_up` para configurar un *pull-up* o un *pull-down*.

```
class InputDevice(GPIODevice):
    def __init__(self, pin=None, pull_up=False):
        ...

    @property
    def pull_up(self):
        return self.pin.pull == 'up'
```

Pero si te has fijado bien no aparecen las propiedades `when_pressed` o `when_released`. ¿Cómo se definen? La respuesta está justo después de la definición de `Button`:

```
Button.is_pressed = Button.is_active
Button.pressed_time = Button.active_time
Button.when_pressed = Button.when_activated
Button.when_released = Button.when_deactivated
Button.wait_for_press = Button.wait_for_active
Button.wait_for_release = Button.wait_for_inactive
```

Añade nombres alternativos, más fáciles de recordar, para las propiedades estándar de todos los dispositivos.

Clases básicas

GPIO Zero tiene un buen montón de clases directamente utilizables:

Clase	Descripción
<code>LED</code>	LED conectado a pin con resistencia a masa.
<code>Buzzer</code>	<i>Buzzer</i> o chicharra conectado a pin directamente con cátodo a masa.
<code>PWMLed</code>	LED conectado a pin con resistencia a masa, con intensidad variable.
<code>RGBLED</code>	LED tricolor a tres pines con resistencia a masa.
<code>Motor</code>	Motor conectado con un puente H a dos pines para movimiento bidireccional.
<code>Button</code>	Pulsador conectado a masa y al pin.
<code>LineSensor</code>	Sensor infrarrojo de línea para robot seguidor de línea.
<code>MotionSensor</code>	Sensor PIR con <i>OUT</i> conectado a pin a través de un divisor de tensión o <i>level converter</i> .
<code>LightSensor</code>	LDR a 3.3V con condensador de 1uF a masa y ambas patas a un pin.
<code>DistanceSensor</code>	Sensor de ultrasonidos con <i>TRIG</i> a un pin y <i>ECHO</i> a otro pin a través de un divisor de tensión o <i>level converter</i> .
<code>MCP3xxx</code>	Convertidores AD de Microchip MCP300x, MCP320x y MCP330x.

Actualmente GPIO Zero no soporta el ADS1118, pero se trata de un dispositivo SPI, como los MCP3xxx, así que su incorporación es sencilla. Lo veremos en seguida.

Además GPIO Zero incluye una clase `CompositeDevice` que permite agrupar varios dispositivos individuales. Tienes varios ejemplos en el archivo `boards.py`.

Ejemplos de uso

Poco más hay que decir de las entradas y salidas digitales, pero para ilustrar su uso vamos a poner los mismos ejemplos que en C.

En C poníamos un ejemplo en que el pin 18 conmutaba cada segundo. En Python podría traducirse directamente así:

```
from gpiozero import LED
from time import sleep

led = LED(18)
while True:
    led.toggle()
    sleep(1)
```

Pero todavía hay una forma más fácil:

```
from gpiozero import LED
from signal import pause

led = LED(18)
led.blink()
pause()
```

Si queremos cambiar el periodo no hay más que pasar parámetros adicionales. Por ejemplo:

```
led.blink(on_time=0.3, off_time=0.7)
```

El uso de PWM es también trivial. Por ejemplo, consideremos el caso de un parpadeo de un LED variando la intensidad de forma sinusoidal, como el LED de los MacBook cuando están en *stand-by*:

```
led.source = scaled(sin_values(100), 0, 1, -1, 1)
```

Al definir la propiedad *source* empieza un hilo que lee valores de esta fuente cada 0.01 segundos por defecto. La función `sin_values` va produciendo valores de un seno entre -1 y 1 con un periodo de 100 muestras. Simplemente tenemos que escalar estos valores que van de -1 a 1 en los correspondientes valores entre 0 y 1. Esa adaptación la hace el generador `scaled`. Todo este tipo de adaptadores que están en el módulo `gpiozero.tools` son extremadamente útiles, vamos a resumir los más importantes.

Función	Descripción
<code>negated</code>	Para cada valor v del iterable devuelve <i>not</i> v .
<code>inverted</code>	Para cada valor v del iterable devuelve $1 - v$.
<code>scaled</code>	Escalado lineal de los valores del iterable.
<code>clamped</code>	Recortado por arriba y por abajo.
<code>absoluted</code>	Para cada valor v del iterable devuelve $abs(v)$.
<code>quantized</code>	Discretiza en un número de pasos homogéneos.
<code>all_values</code>	Genera <i>True</i> cuando todos los iterables generan <i>True</i> .
<code>any_values</code>	Genera <i>True</i> cuando alguno de los iterables generan <i>True</i> .
<code>averaged</code>	Promedia los iterables.
<code>queued</code>	Encola valores y hasta que no se llena la cola no genera.
<code>pre_delayed</code>	Antes de generar espera un tiempo.
<code>post_delayed</code>	Después de generar espera un tiempo.
<code>random_values</code>	Genera números aleatorios entre 0 y 1.
<code>sin_values</code>	Genera valores del seno. Se indica el periodo en muestras.
<code>cos_values</code>	Genera valores del coseno. Se indica el periodo en muestras.
<code>izip</code>	Combina varios generadores generando tuplas de elementos (está en la biblioteca <code>itertools</code>).
<code>cycle</code>	Genera una secuencia infinita a partir de un iterable finito repitiéndolo (está en la biblioteca <code>itertools</code>).

Programación de la interfaz I2C

Desde Python la comunicación con los dispositivos I2C se puede realizar de tres formas:

- La biblioteca *smbus* proporciona acceso desde Python a la funcionalidad del driver I2C de Linux.
- La biblioteca *wiringPi2* que no es sino un envoltorio de la biblioteca C.
- La biblioteca *pigpio* que también ofrece un envoltorio a la versión remota (a través de *pigpiod*) de la biblioteca C.

Actualmente GPIO Zero no proporciona ningún ejemplo de dispositivo I2C. Eso no significa que no pueda integrarse de forma relativamente sencilla, pero los detalles de la programación I2C no se abstraen en modo alguno.

Veamos los detalles de la comunicación con *smbus* y ya nos ocuparemos de la construcción de abstracciones más adelante.

I2C es un protocolo que se utiliza en multitud de dispositivos en ordenadores convencionales. Las placas bases actuales disponen de un *System Management Bus* que utiliza el protocolo I2C para comunicar con los sensores de temperatura, y de voltajes, gestionar la velocidad de los ventiladores, etc. El kernel Linux incorpora un driver que no es específico para Raspberry Pi y es precisamente esta interfaz la que explota la biblioteca *smbus*.

El uso de la biblioteca es sumamente sencillo:

```
from smbus import SMBus

i2c = SMBus(1)
```

Solo el bus 1 está disponible en la Raspberry Pi 3. El bus 0 está reservado para el VideoCore, pero si vamos a prescindir de su uso y de la cámara CSI es posible habilitarlo en `/boot/config.txt` utilizando la opción `i2c_vc=on`.

Ahora tenemos disponible un conjunto de métodos cuyo significado es muy similar al de las funciones I2C de *pigpio* (ver capítulo 3).

```
from smbus import SMBus
from struct import pack, unpack
from time import sleep

i2c = SMBus(1);
assert 0x68 == i2c.read_byte_data(0x68, 117)
i2c.write_byte_data(0x68, 107, 0)
while True:
    bytes = i2c.read_i2c_block_data(0x68, 59, 14)
    data = unpack('>7h', pack('14B', bytes))
    print("temperatura:", data[3]/340. + 36.53)
    print("ax={}, ay={}, az={}".format(*data))
    sleep(.5)
i2c.close()
```

Método	Significado
<code>open(bus)</code>	Conecta el objeto con un canal I2C para el bus indicado.
<code>close()</code>	Desconecta el objeto del bus I2C.
<code>write_byte_data(addr, reg, v)</code>	Escribe un byte <i>v</i> en un registro <i>reg</i> .
<code>write_word_data(addr, reg, v)</code>	Escribe un short <i>v</i> en un registro <i>reg</i> .
<code>read_byte_data(addr, reg)</code>	Lee un byte de un registro <i>reg</i> .
<code>read_word_data(addr, reg)</code>	Lee un short de un registro <i>reg</i> .
<code>write_i2c_block_data(addr, reg, data)</code>	Escribe un bloque a partir de <i>reg</i> .
<code>read_i2c_block_data(addr, reg, len)</code>	Lee un bloque a partir de <i>reg</i> .

Evidentemente es relativamente sencillo adaptar esta interfaz a la filosofía de GPIO Zero. Queda propuesto como ejercicio traducir el ejemplo anterior en un *InputDevice* cuyos valores son tuplas con los tres datos de aceleración.

Programación en Python de SPI

En Python disponemos de varias bibliotecas para acceder a dispositivos SPI:

- GPIO Zero, que incorpora varios convertidores AD de Microchip.
- La biblioteca *spidev* proporciona acceso desde Python a la funcionalidad del driver SPI de Linux.
- La biblioteca *wiringPi2* que no es sino un envoltorio de la biblioteca C.
- La biblioteca *pigpio* que también ofrece un envoltorio a la versión remota de la biblioteca C.

La biblioteca más sencilla es nuevamente GPIO Zero. Veamos un ejemplo de la documentación de `scaled`:

```
from gpiozero import Motor, MCP3008
from gpiozero.tools import scaled
from signal import pause

motor = Motor(20, 21)
pot = MCP3008(channel=0)
motor.source = scaled(pot.values, -1, 1)
pause()
```

El potenciómetro está conectado al canal 0 de un conversor AD Microchip MCP3008. Se trata de un dispositivo SPI, igual que nuestro ADS1118. El usuario está por tanto completamente aislado de la interacción SPI. Pero vamos a ver lo que implicaría definir un módulo específico para el ADS1118.

La clase *SPIDevice*

Todos los dispositivos SPI derivan de la clase `SPIDevice`. Esta define un atributo privado `self._spi` que se construye llamando a la función `SPI`. Esta función devuelve un `SPIHardwareInterface` o un `SPISoftwareInterface` dependiendo de los pines utilizados. Si los pines corresponden con un puerto SPI de la Raspberry Pi devolverá un `SPIHardwareInterface`, que es mucho más eficiente.

Para cambiar la polaridad del reloj y la fase podemos utilizar la propiedad `clock_mode` del atributo `self._spi`, que en nuestro caso (ADS1118) debe tener el valor 1.

En lugar de usar `SPIDevice` como base podemos aprovechar la clase `AnalogInputDevice`, que añade la lógica necesaria para decodificar el valor escalado al rango [0, 1] dependiendo del número de bits de las lecturas.

Con esto la clase para implementar un dispositivo ADS1118 queda prácticamente igual que los MCP3xxx.

```

class ADS1118(AnalogInputDevice):

    def __init__(self, channel=0, differential=False, **spi_args):
        self._channel = channel
        self._bits = 16
        self._differential = bool(differential)
        super(ADS1118, self).__init__(16, **spi_args)
        self._spi.clock_mode = 1
        data = self._spi.transfer(2 * self._config_reg())

    @property
    def channel(self):
        return self._channel

    @property
    def differential(self):
        return self._differential

    def _read(self):
        data = self._spi.transfer(2 * self._config_reg())
        result = (data[0] << 8) | data[1]
        if self.differential and result > 32767:
            result = -(65536 - result)
        assert -32768 <= result < 32768
        return result

    def _config_reg(self):
        ''' Configuración en modo continuo a 128 SPS y con rango a
            plena escala de +-2.048V'''
        #   Byte      0      1
        #   ====
        #           sMCCGGGS rrrTP011
        #
        #   s = start single shot conversion
        #   M = differential (1 = single-ended, 0 = differential)
        #   C = channel
        #   G = gain
        #   S = single shot mode (1 = single shot, 0 = continuous)
        #   r = sample rate
        #   T = temperature sensor
        #   P = pull-up in DOUT
        return [0b00000100 + (self.channel << 4) + [0b01000000, 0][self.differential],
                0b10001011]

```

Ya está, no hay más, con esto ya se puede usar como cualquiera de los módulos AD incluidos en GPIO Zero. Conecta un potenciómetro entre 3.3V y masa y el cursor utilízalo como entrada AN0 del módulo CJMCU-1118. Conecta el módulo a la interfaz SPI, usando como línea de selección CE0.

```

led = PWMLED(4)
pot = ADS1118(channel=0)
led.source = pot.values

```

Un LED con intensidad controlada por un potenciómetro.

Comunicaciones en red

En el capítulo 2 ya hemos introducido la interfaz *socket*. En Python se incluye esta interfaz en la biblioteca `socket`. La forma de usar esta biblioteca es prácticamente lo que hemos visto en el capítulo 2, con la única diferencia de que se usa `send` en lugar de `write` y `recv` en lugar de `read`.

Comunicaciones UDP

El siguiente ejemplo muestra el servidor UDP más simple posible.

```
from socket import *

s = socket(AF_INET, SOCK_DGRAM)
s.bind(('0.0.0.0', 8888))
while True:
    data = s.recv(1024)
    if not data:
        break
    print(data)
```

Al construir un *socket* Python devuelve un objeto que implementa toda la *API socket*. En UDP lo único que tenemos que hacer es asignar una dirección y puerto de escucha y a partir de ese momento empezamos a recibir o enviar mensajes indicando un tamaño de buffer. Recuerda que en Python se lee con `recv` (o `recvfrom`).

El cliente correspondiente es aún más simple.

```
from socket import *

s = socket(AF_INET, SOCK_DGRAM)
s.connect(('localhost', 8888))
while True:
    s.send(input().encode('utf-8'))
```

Para establecer la dirección destino usamos `connect` y desde ese momento ya podemos enviar mensajes con `send`.

Comunicaciones TCP

Desde el punto de vista de la programación la diferencia fundamental con el caso anterior radica en la necesidad de establecer conexiones para realizar una comunicación. Esto complica algo el servidor.

```
from socket import *

s = socket(AF_INET, SOCK_STREAM)
s.bind(('0.0.0.0', 8888))
s.listen(10)

con, addr = s.accept()
while True:
    data = con.recv(1024)
    if not data:
        break
    print(data)
con.close()
```

Al igual que antes asignamos la dirección y puerto en los que escucha, pero ahora tenemos que configurar el *backlog* con `listen` (cuántas conexiones se retienen sin aceptar) y aceptar nuevas conexiones con `accept`. El método `accept` devuelve otro *socket* que usamos para el diálogo con ese cliente conectado.

En realidad esto es una sobre-simplificación. Una vez aceptada la conexión se puede atender en un hilo independiente y aceptar nuevas conexiones en el hilo principal inmediatamente.

El cliente es prácticamente idéntico al de UDP.

```
from socket import *  
  
s = socket(AF_INET, SOCK_STREAM)  
s.connect(('localhost', 8888))  
while True:  
    s.send(input().encode('utf-8'))
```


Arquitectura de software

La arquitectura de un programa no es más que su organización interna, los componentes en los que se divide y la relación entre ellos. Si buscas bibliografía sobre el tema verás centenares de libros que hablan de la arquitectura de software como una evolución de la artesanía a la ingeniería en el campo del software. No leerás en este manual nada sobre esto, porque sigo pensando que la programación es una labor artesanal, como el trabajo de los mejores ingenieros. Y como tal tiene que tener un componente estético muy importante. Valorar y entrenar el arte de programar es una parte importante de la profesionalización de la actividad.

Podemos escribir decenas de páginas sobre requisitos, diseño arquitectónico, diseño detallado, etc. Pero nada de esto podrá despertar el gusanillo de los *hackers* originales, los apasionados por la tecnología, que hacían de la mejora continua su propia forma de vida.

En este capítulo te ofrecemos una posible forma de organizar tu código. Ni es la única posible ni posiblemente sea la mejor. Cumple con su función de equilibrar la eficiencia con la simplicidad y la facilidad de uso por programadores principiantes. Pero si tienes sugerencias sobre posibles mejoras no te las guardes, envíanos realimentación para mejorar las próximas ediciones.

A diferencia de la versión C en Python tenemos soporte del lenguaje para todas las técnicas básicas. Programación orientada a objetos con clases y excepciones son elementos que vamos a usar extensivamente. El soporte nativo de constructores y destructores en el lenguaje abre la puerta de una técnica muy importante en el software actual: RAI (*Resource Acquisition Is Initialization*) que usaremos extensivamente. Sin embargo no necesitamos contar todo esto, porque hay muchos libros libres que lo cuentan. Usa tu libro favorito de Python si hay algún detalle que no entiendes.

Apéndices

En este capítulo aglutinamos secciones de interés que no se impartirán en el curso. La mayoría están aquí por motivos históricos y es posible que se renueven por completo o se eliminen para la próxima edición.

Nuestra personalización de Raspbian

Este capítulo describe todas las acciones de personalización que se han realizado en la tarjeta microSD que se distribuye en el taller.

- Las tarjetas están ya formateadas en VFAT por el fabricante. No se ha realizado ningún formateo adicional.
- Partimos de la versión de NOOBS 1.9 descargada del [sitio oficial](#). El archivo `zip` se ha descomprimido íntegramente en la tarjeta vacía.
- En el primer arranque aparece el menú de NOOBS con una primera opción de instalación, **Raspbian**. Se instala de forma automática.
- Configurar la disposición (*layout*) de teclado en castellano por defecto.
- Ejecutar la aplicación de configuración y en la pestaña de *Interfaces* habilitar SPI, I2C y Serial. También en la pestaña de *Localisation* seleccionar *Locale* `es/ES UTF-8`. Además seleccionar la zona horaria (*Timezone*) `Europe/Madrid`, el teclado como `Spain/Spanish` y finalmente la zona WiFi como `ES/Spain`.
- Al reiniciar, el sistema entra automáticamente en modo gráfico. Configurar los iconos para lanzamiento rápido de aplicaciones de manera que tengamos acceso rápido a la aplicación de configuración de la Raspberry Pi, un terminal, un navegador, un editor de textos y el entorno de programación IDLE.
- Configurar el fondo de escritorio al archivo `rpi-uc1m.svg`.
- Instalar algunos paquetes adicionales: `tmux`, `screen`, `bc`, `i2c-tools`, `python-smbus`, `python3-smbus`, `ipython`, `ipython3`, `zile`, `python-dev`, `python-gpiozero`, `python3-dev`, `python3-gpiozero`, `mpg123`, `manpages-es`, `gcc-4.9-doc`, `gdb-doc`, `wireshark`, `liblo-dev`, `python-liblo`, `python3-liblo`.
- Añadir el usuario `pi` al grupo `staff` para que funcione `pip install`, y al grupo `wireshark` para que podamos capturar tráfico sin ser superusuario.
- Instalar la biblioteca *wiringPi* para Python con `pip install wiringpi2` y `pip3 install wiringpi2`.
- Instalar la biblioteca *bcm2835* con `wget http://www.airspayce.com/mikem/bcm2835/bcm2835-1.50.tar.gz`, `tar xzvf bcm2835*tar.gz`, `cd bcm2835-1.50`; `./configure`; `make`; `sudo make install`.
- Instalar la biblioteca *pigpio* con `wget https://github.com/joan2937/pigpio/archive/master.zip`, `unzip master.zip`, `cd pigpio-master`; `make`; `sudo make install`.
- Instalar `pygubu` (editor de interfaces gráficas) con `sudo pip install pygubu` y `sudo pip3 install pygubu`.
- Descargar las pruebas del sistema del repositorio GitHub del taller con `git clone https://github.com/FranciscoMoya/rpi-src.git src` y `git clone https://github.com/FranciscoMoya/rpi-doc.git doc`.
- Instalar `nodejs`, `npm` y `node-semver`.
- Instalar `gitbook` con `sudo npm install gitbook-cli -g`.
- Compilar la documentación con `gitbook install`, `gitbook build` en la carpeta `~/doc`.
- Actualizar firmware con `rpi-update`.

Secuencia de arranque

Una vez que la Raspberry Pi recibe alimentación se inicia una secuencia de operaciones que se conoce como *secuencia de arranque*. Está básicamente documentada en elinux.org:

- Cuando la Raspberry Pi se enciende el procesador ARM está apagado, y la GPU VideoCore IV está encendida. En este punto incluso la propia memoria SDRAM está deshabilitada.
- El pequeño núcleo RISC de la GPU empieza a ejecutar la primera etapa del *bootloader* que está almacenada en la ROM del BCM2835. Esta primera etapa lee la tarjeta microSD, que debe tener una primera partición en formato FAT32, y carga la segunda etapa del *bootloader*, que corresponde al contenido del archivo `bootcode.bin`. Puesto que la memoria SDRAM sigue deshabilitada esta carga se produce en la memoria cache de segundo nivel, y la ejecuta.
- El código de `bootcode.bin` habilita la SDRAM y lee el archivo `start.elf` que contiene el firmware de la GPU en formato ELF (el típico de los binarios en GNU/Linux). Este archivo es también ejecutado por la GPU. Entre otras cosas este archivo contiene los codecs de video y la implementación de OpenGL ES.
- El código de `start.elf` arranca la CPU ARM. Un archivo adicional `fixup.dat` se usa para configurar la partición de la SDRAM entre la GPU y la CPU. A partir de aquí el sistema operativo contenido en el archivo `kernel.img` se ejecuta en la CPU ARM.
- El archivo `kernel.img` es el primer archivo de la secuencia de arranque que ejecuta el procesador ARM. Habitualmente contendrá el núcleo del sistema operativo pero puede contener programas arbitrarios. Como ejemplos puede consultarse [el repositorio GitHub de David Welch](#).

Esta secuencia de arranque implica que la Raspberry Pi requiere de una tarjeta microSD para poder arrancar. Sin embargo esto tiene una ventaja clara, es imposible inutilizarla por errores de programación. En la jerga habitual en sistemas empujados se dice que no puede ser enladrillada (*bricked* en inglés).

La primera etapa del *bootloader* solo sabe leer particiones formateadas en FAT32. Esto es especialmente relevante con las tarjetas microSD de gran tamaño, que habitualmente vienen formateadas en exFAT y Microsoft Windows 8 y posteriores también utilizan por defecto exFAT para formatearlas. Hay multitud de mensajes en los foros alertando de tarjetas que no funcionan con Raspberry Pi cuando lo único que pasa es que no están adecuadamente formateadas.

Alimentación de Raspberry Pi

Todos los modelos de Raspberry Pi con la única excepción del *Compute Module* reciben la alimentación a través de un conector microUSB independiente del que se utiliza para la conexión de periféricos. Esto garantiza que no podemos meter la pata en la alimentación, todos los alimentadores microUSB son seguros porque proporcionan una tensión de 5V con la polaridad adecuada.

Con un modelo A o B original, para evitar problemas con los dispositivos USB (especialmente cuando se usan interfaces de red WiFi o Ethernet, o discos duros portátiles) es necesario utilizar un *hub USB* alimentado externamente.

Es más, muchos de los hubs USB disponibles en el mercado (especialmente los más baratos) permiten alimentar la Raspberry Pi directamente desde el puerto USB al que se conecta (*upstream port*), dejando desconectado el puerto de alimentación. Es preciso aclarar que esta característica conocida como retro-alimentación (*back powering*) hace que la alimentación no pase por el circuito de protección contra sobretensiones. Por tanto es muy importante que el hub empleado tenga **protección contra sobre-corriente con un límite no superior a 2.5A**. Afortunadamente casi todos los circuitos integrados de los hub USB incorporan el mecanismo de detección de sobre-corriente en el propio chip, por lo que es bastante probable que el hub implemente dicha protección.

Ha habido una amplia polémica en los foros acerca de qué formas de alimentación son las ideales. Se han propuesto otras alternativas, como la de **utilizar el conector de GPIO para alimentar el circuito conectando directamente una fuente a los pines +Vcc y GND**. Pero nuevamente se salta el fusible de protección, por lo que debemos asegurar que se implementan las debidas protecciones contra sobre-corriente.

Si investigas un poco este tema verás que hay una amplia variedad de comentarios que parecen prevenir contra el *back powering* por saltarse el fusible de protección. Sin embargo ten presente que lo que se pretende proteger es una Raspberry Pi de menos de 30€. En el caso de la Raspberry Pi Zero no llega a 5€. Si el hub no implementa las protecciones necesarias es verdad que se pondría en riesgo la Raspberry Pi, pero también se pondría en riesgo cualquier otro dispositivo que se conecte en los puertos *downstream*. En particular se pondría en riesgo los discos duros, las cámaras, etc. La mayoría de ellos tienen un coste comparable o superior a la Raspberry Pi y nadie cuestiona que se conecten al hub. A fin de cuentas está para eso.

Si tu hub alimentado no permite *back powering* la Raspberry Pi no se encenderá al conectarla al hub. En ese caso es todavía mejor, es lo correcto según la especificación de USB. Simplemente hay que conectar un cable de uno de los puertos del hub al puerto microUSB de alimentación. Ocupamos un puerto más pero estaremos seguros de que el circuito está protegido contra sobrecorrientes.

Los modelos A+, B+, 2B y 3B ya tienen un *hub USB* incorporado y el fusible de protección tiene una corriente nominal sensiblemente superior (2A), por lo que se suele alimentar con un alimentador microUSB de 2A normal, como los que se usan para cargar los teléfonos modernos. En el caso del modelo 3B es necesario disponer de un alimentador de 2.5A.

El circuito de alimentación de la Raspberry Pi no es muy tolerante a tensiones de alimentación por debajo de la nominal, pero todos los modelos a partir del B+ incorporan un mecanismo de detección de tensión de alimentación demasiado baja o de sobrecalentamiento.

- Cuando la tensión de alimentación cae por debajo de 4.65V se apaga el LED de alimentación y se muestra un pequeño cuadradito de colores en la esquina superior derecha de la pantalla.
- Cuando la temperatura sube por encima de 85°C se muestra un pequeño cuadradito rojo en la esquina superior derecha de la pantalla.

Hemos detectado en múltiples ocasiones la condición de *under-voltage* al emplear alimentadores de muy bajo precio en los modelos 2B y 3B, en los que el consumo es significativamente superior a los basados en BCM2835. También hemos detectado la condición de *under-voltage* al usar cables microUSB demasiado largos o de baja calidad, y al conectar periféricos USB junto a un conversor HDMI-VGA.

Por este motivo en el *kit del alumno* empleamos fuentes de alimentación oficiales diseñadas por la *Raspberry Pi Foundation*. Se trata de diseños de alta eficiencia ligeramente sobredimensionados. Generan una tensión de alimentación de 5.1V que está dentro de los límites tolerados y compensa la posible caída de tensión cuando aumenta la carga del sistema (p.ej. en el arranque).

La fuente de alimentación de la Raspberry Pi

La alimentación externa de la Raspberry Pi se utiliza para generar las diferentes tensiones de alimentación que requieren. La fuente de alimentación de la Raspberry Pi genera tensiones de 5V, 3.3V, 2.5V y 1.8V que se necesitan en varias partes del PCB.

Esta parte del circuito de la Raspberry Pi es la que ha experimentado una evolución más significativa. Una amplia discusión sobre las diferencias puede consultarse en el [blog de Adafruit](#).

Los modelos iniciales empleaban reguladores lineales mientras que los modelos B+ y posteriores emplean fuentes conmutadas mucho más eficientes. La diferencia en consumo puede ser notable.

Además es frecuente leer en los foros que los modelos originales se resetean al desconectar dispositivos USB. Con los modelos A+ y B+ se añade un circuito protector para la alimentación especialmente diseñado para permitir las conexiones y desconexiones en caliente. En cambio, a partir de la revisión 2 del modelo original ya no se incluye limitación de corriente de alimentación.

<https://es.scribd.com/doc/101830961/GPIO-Pads-Control2>

Parámetros configurables

Todos los parámetros de configuración inicial de la Raspberry Pi se buscan en un archivo opcional denominado `config.txt` que debe estar en la misma partición FAT32 que el *bootloader*. Este fichero es leído por la GPU antes de arrancar la CPU y puede verse como el equivalente a las memorias Flash o EEPROM empleadas por el BIOS de los computadores personales para guardar los parámetros de inicialización.

El formato del fichero es extremadamente simple. Consiste en líneas del tipo `parámetro=valor` o líneas de comentarios que empiezan por el signo `#`. Entre los parámetros configurables se encuentran los diferentes modos de video y su configuración, la partición de la memoria cache entre GPU y CPU, la posibilidad de deshabilitar la memoria cache de nivel 2, la posibilidad de elevar la frecuencia de diversos componentes por encima de las especificaciones del fabricante (*overclocking*), los códigos de activación de los codecs no libres, o los parámetros de arranque del sistema operativo. Una detallada explicación de la mayoría de estos parámetros puede encontrarse en elinux.org.

Actualmente la distribución de software *Raspbian*, que es la que usamos en el taller, ya dispone de una herramienta de configuración, que no hace sino editar de forma interactiva este archivo.

Actualmente los detalles de configuración están plenamente documentados por la Raspberry Pi Foundation en raspberrypi.org.

Referencias