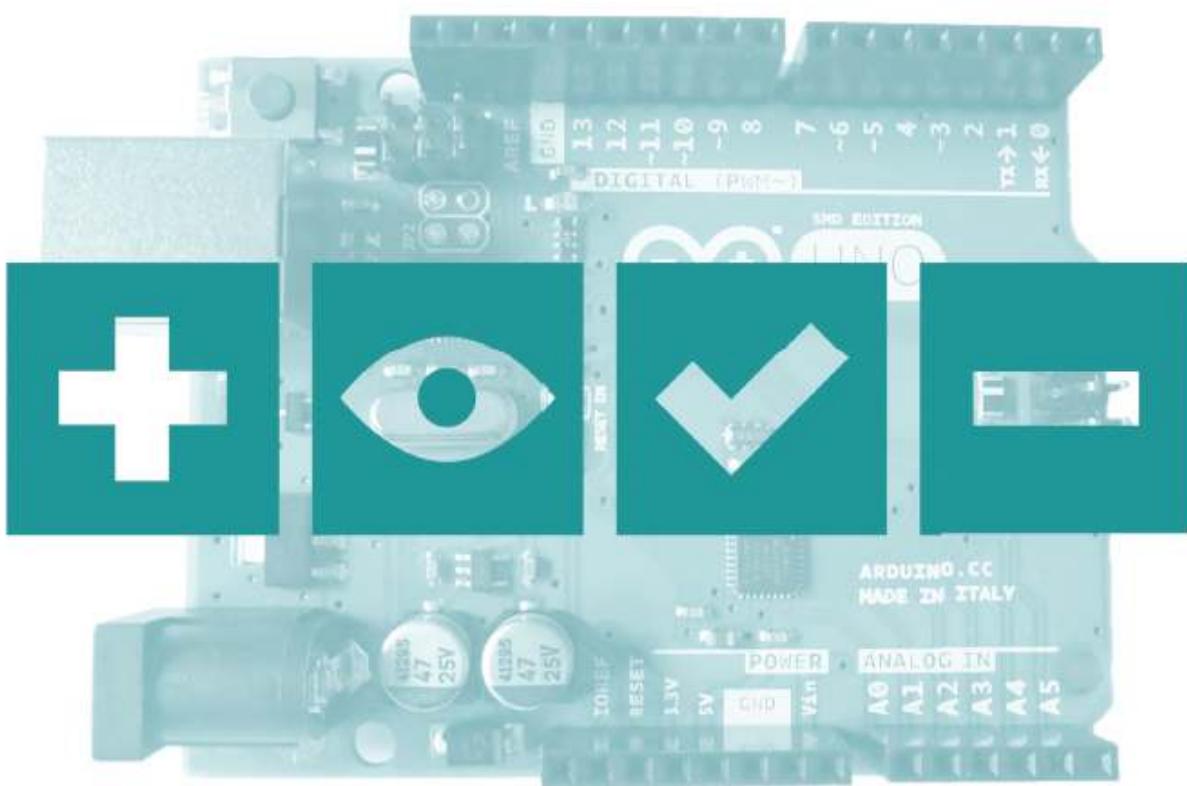


El mundo

GENUINO-ARDUINO

Curso práctico de formación



Óscar Torrente Artero

ΔΔ Alfaomega



El mundo

GENUINO-ARDUINO

Curso práctico de formación

El mundo
GENUINO-ARDUINO
Curso práctico de formación

Óscar Torrente Artero



Diseño de colección, cubierta
y pre impresión: Grupo RC

Datos catalográficos

Torrente, Óscar
El mundo GENUINO-ARDUINO. Curso práctico de
formación
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México

ISBN: 978-607-622-641-4

Formato: 17 x 23 cm

Páginas: 568

El mundo GENUINO-ARDUINO. Curso práctico de formación

Óscar Torrente Artero

ISBN: 978-84-943450-2-9 edición original publicada por RC Libros, Madrid, España.

Derechos reservados © 2016 RC Libros

Primera edición: Alfaomega Grupo Editor, México, Febrero 2016

© 2016 Alfaomega Grupo Editor, S.A. de C.V.

Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana

Registro No. 2317

Pág. Web: <http://www.alfaomega.com.mx>

E-mail: atencionalcliente@alfaomega.com.mx

ISBN: 978-607-622-641-4

Derechos reservados:

Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Nota importante:

La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele.

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Pitágoras 1139, Col. Del Valle, México, D.F. – C.P. 03100.
Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia,
Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Paraguay 1307 P.B. Of. 11, C.P. 1057, Buenos Aires,
Argentina, – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaditor.com.ar

A mi madre

Los esquemas eléctricos han sido realizados con CircuitLab: <http://www.circuitlab.com>

Los gráficos de circuitos han sido realizados con Fritzing: <http://www.fritzing.org>

Los retoques han sido realizados con Inkscape y Gimp: <http://inkscape.org>, <http://gimp.org>

Las imágenes han sido obtenidas por medios propios o bien descargadas de la Wikipedia ó Ladyada.net (con licencia CC-Share-Alike): <http://es.wikipedia.org>,<http://www.ladyada.net>

ÍNDICE

INTRODUCCIÓN.....	XV
CAPÍTULO 1. ELECTRÓNICA BÁSICA.....	1
CONCEPTOS TEÓRICOS SOBRE ELECTRICIDAD.....	1
¿Qué es la electricidad?	1
¿Qué es el voltaje?	2
¿Qué es la intensidad de corriente?	3
¿Qué es la corriente continua (DC) y la corriente alterna (AC)?	4
¿Qué es la resistencia eléctrica?	5
¿Qué es la Ley de Ohm?.....	5
¿Qué es la potencia?	6
¿Qué son las señales digitales y las señales analógicas?	7
¿Qué son las señales periódicas y las señales aperiódicas?.....	9
CIRCUITOS ELÉCTRICOS BÁSICOS.....	11
Representación gráfica de circuitos	11
Circuitos abiertos, cerrados y cortocircuitos.....	12
Conexiones en serie y en paralelo	13
El divisor de tensión	16
Las resistencias "pull-up" y "pull-down"	17
FUENTES DE ALIMENTACIÓN ELÉCTRICA.....	19
Tipos de pilas/baterías	19
Voltaje de corte, capacidad y capacidad de las pilas/baterías.....	22
Conexiones de varias pilas/baterías.....	24
Compra de pilas/baterías	25
Compra de portapilas (con distintos conectores)	26
Compra de cargadores	28
Breve nota sobre los conectores y el protocolo USB.....	30
Características de los adaptadores AC/DC	31
Breve nota sobre las fuentes de alimentación solares.....	35
COMPONENTES ELÉCTRICOS.....	36
Resistencias.....	36
Potenciómetros.....	39

EL MUNDO GENUINO-ARDUINO

Breve nota sobre los "softpots" o potenciómetros de "membrana"	41
Otras resistencias de valor variable	42
Diodos (y LEDs)	42
Breve nota sobre los "datasheets"	43
Un tipo de diodo muy particular: el LED	45
Breve nota sobre los LEDs RGB	47
Condensadores	48
Usos comunes de los condensadores: desacople y filtro	51
Transistores	52
Pulsadores	56
Otros tipos de interruptores (o conmutadores)	57
Reguladores de tensión	59
Breve nota sobre los elevadores DC/DC	62
Placas de prototipado	64
Cables	68
USO DE UNA PLACA DE PROTOTIPADO.....	70
Breve nota sobre cómo alimentar circuitos en placas de prototipado	70
USO DE UN MULTÍMETRO DIGITAL.....	77
CAPÍTULO 2. HARDWARE GENUINO	83
¿QUÉ ES UN SISTEMA ELECTRÓNICO?.....	83
¿QUÉ ES UN MICROCONTROLADOR?.....	84
¿QUÉ ES GENUINO/ARDUINO?	86
¿CUÁL ES EL ORIGEN DE ARDUINO?.....	88
¿QUÉ QUIERE DECIR QUE ARDUINO SEA "SOFTWARE LIBRE"?	89
¿QUÉ QUIERE DECIR QUE ARDUINO SEA "HARDWARE LIBRE"?	90
¿POR QUÉ ELEGIR ARDUINO?	92
EL MICRO DE LAS PLACAS ARDUINO (y del modelo UNO en particular)	93
El encapsulado del microcontrolador	93
DIP	94
SMD	95
El modelo del microcontrolador	97
Arquitectura AVR	98
Arquitectura ARM	98
Breve nota sobre AVR vs. ARM (y x86).....	99
El chip ATmega328P	100
Las memorias del microcontrolador	101
Breve nota sobre las unidades de medida de la información	102
Breve nota sobre las diferencias entre memorias Flash y EEPROM.....	105
Los registros del microcontrolador	106
La comunicación serie con el exterior	106

ÍNDICE

Comunicación asíncrona	108
Comunicación síncrona	108
El gestor de arranque ("bootloader") del microcontrolador.....	113
Los gestores de arranque de las placas Due y Zero (ARM).....	115
Otros gestores de arranque más exóticos	116
CARACTERÍSTICAS DE LA PLACA ARDUINO UNO	117
La alimentación eléctrica	117
El chip ATmega16U2.....	121
Breve nota sobre la tecnología TTL y sus niveles HIGH/LOW aceptados	121
Reprogramación del chip ATmega16U2.....	123
Breve nota sobre los "pogo pins"	124
Breve nota sobre los VID y PID	126
Las entradas y salidas digitales	126
Las entradas analógicas	127
Las salidas analógicas (PWM)	129
Otros usos de los pines-hembra de la placa	133
El conector ICSP	136
SPI.....	136
ISP.....	137
Breve nota sobre cómo realizar una programación ISP con el entorno Arduino	140
El reloj	141
Los temporizadores ("timers") del microcontrolador	143
El botón de "reset".....	144
Obtener el diseño esquemático y de referencia	145
¿QUÉ OTRAS PLACAS ARDUINO OFICIALES EXISTEN?	146
Arduino Pro	146
Los pines-hembra (y otros).....	147
Los adaptadores USB-Serie	148
Arduino Pro Mini.....	149
Arduino Nano.....	150
Arduino Mega 2560	151
Arduino Micro	151
El "auto-reset" del micro ATmega32U4.....	153
Arduino Yún.....	153
Breve nota sobre cómo conseguir cargar programas en la placa Yún vía WiFi	157
Arduino Lilypad, Lilypad Simple, Lilypad SimpleSnap y LilypadUSB.....	161
Arduino Gemma.....	163
Arduino Due	163
Arduino Zero	165
Arduino 101	167
Tablas comparativas de los diferentes modelos de placas.....	168
Breve nota sobre las regulaciones del espectro electromagnético.....	170
¿QUÉ "SHIELDS" ARDUINO OFICIALES EXISTEN?.....	170

EL MUNDO GENUINO-ARDUINO

Arduino Ethernet Shield	171
Breve nota sobre Ethernet.....	172
PoE ("Power Over Ethernet").....	174
Arduino WiFi Shield 101	177
Arduino GSM Shield.....	178
Arduino Motor Shield	179
Arduino Proto Shield.....	181
¿QUÉ SHIELDS NO OFICIALES EXISTEN?	182
Proto Shields.....	182
Power Shields	183
¿QUÉ PLACAS ARDUINO NO OFICIALES EXISTEN?	184
CAPÍTULO 3. SOFTWARE ARDUINO.....	191
¿QUÉ ES UN IDE?.....	191
INSTALACIÓN DEL IDE ARDUINO	192
Cualquier sistema Linux	192
Posible problema: la configuración del gestor de ficheros Nautilus	193
Posible problema: los permisos de usuario	193
Breve nota sobre el reconocimiento y uso de dispositivos USB-ACM en Linux	194
Cualquier sistema Linux (a partir del código fuente).....	195
Windows	195
Posible problema: instalación del "driver"	196
Breve nota sobre el reconocimiento y uso de dispositivos COM en Windows	196
OS X	196
PRIMER CONTACTO CON EL IDE	197
HERRAMIENTAS EXTRA INTEGRADAS EN EL IDE	203
Las librerías y el "Library Manager"	203
Concepto de librería	203
Cómo instalar librerías (de terceros) manualmente	204
Cómo instalar librerías (de terceros) usando el "Library Manager"	205
Cómo importar librerías.....	206
El "Boards Manager"	207
El "Serial Monitor" y otros terminales serie	210
Ejecución del "auto-reset" al abrir el "Serial Monitor"	211
COMPROBACIÓN DEL CORRECTO FUNCIONAMIENTO DEL IDE	213
USO DEL IDE EN EL INTÉRPRETE DE COMANDOS	215
USO DEL IDE "ARDUINO CREATE".....	217
OTROS IDEs ALTERNATIVOS	218
Entornos "online"	220
Entornos de programación gráfica.....	221
MÁS ALLÁ DEL LENGUAJE ARDUINO: EL LENGUAJE C/C++.....	223

ÍNDICE

Herramientas de compilación C/C++ y carga incluidas en el IDE	225
Herramientas invocadas mediante el botón "Verify"	225
Herramientas invocadas mediante el botón "Upload"	227
CAPÍTULO 4. LENGUAJE ARDUINO	229
MI PRIMER SKETCH ARDUINO	229
ESTRUCTURA GENERAL DE UN SKETCH	230
Sobre las mayúsculas, tabulaciones y los punto y coma	231
COMENTARIOS	232
VARIABLES	233
Declaración e inicialización de una variable.....	233
Asignación de valores a una variable	234
Ámbito de una variable.....	235
Tipos posibles de una variable	236
Tipos simples.....	236
Breve nota sobre ASCII, ISO-8859-1 y UTF-8	238
Breve nota sobre el uso de los sistemas binario y hexadecimal.....	240
Breve nota sobre la importancia de los rangos de valores válidos.....	241
Tipos complejos.....	244
Breve nota sobre los arrays de caracteres y el tipo de datos <i>String</i>	246
Breve nota sobre los punteros	247
La instrucción <i>sizeof()</i>	250
Cambio de tipo de datos (numéricos).....	251
CONSTANTES	254
PARÁMETROS DE UNA INSTRUCCIÓN	255
VALOR DE RETORNO DE UNA INSTRUCCIÓN	256
LA COMUNICACIÓN SERIE CON LA PLACA ARDUINO	257
Instrucciones para enviar datos desde la placa al exterior	258
Uso del "Serial Plotter"	262
Instrucciones para recibir datos desde el exterior	263
Los objetos serie de otras placas Arduino diferentes de la UNO	271
INSTRUCCIONES DE GESTIÓN DEL TIEMPO	273
INSTRUCCIONES MATEMÁTICAS, TRIGONOMÉTRICAS Y DE PSEUDOALEATORIEDAD	275
INSTRUCCIONES DE GESTIÓN DE CADENAS	282
CREACIÓN DE INSTRUCCIONES (FUNCIONES) PROPIAS.....	287
Funciones con parámetros opcionales ("sobrecarga")	291
Funciones con estructuras como parámetros o valor de retorno	292
Funciones con más de un valor de retorno ("paso por referencia")	294
Las variables <i>static</i>	296

EL MUNDO GENUINO-ARDUINO

BLOQUES CONDICIONALES.....	297
Los bloques <i>if</i> e <i>if/else</i>	297
El bloque <i>switch</i>	303
BLOQUES REPETITIVOS (BUCLAS)	305
El bloque <i>while</i>	305
El bloque <i>do</i>	308
El bloque <i>for</i>	308
Las instrucciones <i>break</i> y <i>continue</i>	312
CAPÍTULO 5. LIBRERÍAS ARDUINO	315
LAS LIBRERÍAS OFICIALES	315
Librería LiquidCrystal	316
Librería SD.....	316
Librería Ethernet.....	316
Librería WiFi101.....	317
Librería Temboo	317
Librería GSM	317
Librería SPI	317
Librería Wire	318
Librería SoftwareSerial	318
Librería Firmata	319
Librerías Servo y Stepper	320
Librerías Keyboard y Mouse (solo para placas basadas en el chip ATmega32U4 y para los modelos Due y Zero)	320
Librería EEPROM (para todas las placas excepto los modelos Due y Zero)	321
Librerías USBHost y Scheduler (solo para los modelos Due y Zero)	321
Librería Audio (solo para el modelo Due)	322
Librerías AudioZero y RTCZero (solo para el modelo Zero).....	322
Librerías Bridge y SpacebrewYún (solo para el modelo Yún).....	323
USO DE PANTALLAS LCD	324
Las pantallas de cristal líquido (LCDs).....	324
La librería LiquidCrystal	327
Librerías de terceros interesantes para usar con LCDs	334
Módulos LCD de tipo I ² C o TTL-Serie	335
Backpacks I ² C	336
Backpacks serie	337
Shields que incorporan LCDs	339
Shields y módulos que incorporan GLCDs	341
Breve nota sobre los convertidores de nivel bidireccionales.....	343
Breve nota sobre la visualización de imágenes "al vuelo"	345
USO DE PANTALLAS TFT	347
Shields y módulos que incorporan pantallas TFT	347
Breve nota sobre la librería "Adafruit GFX"	348

ÍNDICE

Shields y módulos que incorporan pantallas TFT táctiles	351
Pantallas TFT táctiles resistivas vs. pantallas TFT táctiles capacitivas	351
De tecnología resistiva	352
De tecnología capacitiva	357
USO DE PANTALLAS OLED.....	359
Las pantallas OLED	359
Módulos OLED de 4DSystems	359
Módulos OLED de Adafruit	360
USO DE OTRAS PANTALLAS.....	362
7-segmentos.....	363
Necesidad de aumentar el número de pines de salida	364
Shields y módulos que incorporan displays 7-segmentos.....	365
Matrices de LEDs.....	369
USO DE LA MEMORIA EEPROM.....	371
USO DE TARJETAS SD	375
Características de las tarjetas SD	375
Shields y módulos que incorporan zócalos microSD.....	377
La librería SD	379
USO DE PUERTOS SERIE SOFTWARE	388
USO DE MOTORES	392
Conceptos básicos sobre motores	392
Tipos de motores	394
Los motores DC	394
Los servomotores	396
Los motores paso a paso	399
La librería Servo	402
La librería Stepper	407
CAPÍTULO 6. ENTRADAS Y SALIDAS.....	411
USO DE LAS ENTRADAS Y SALIDAS DIGITALES	411
Ejemplos con salidas digitales	414
Evitando el uso de la función <i>delay()</i> –y de <i>delayMicroseconds()</i> –.....	418
Múltiples salidas en paralelo	424
Ejemplos con entradas digitales (pulsadores)	437
Implementación de pulsadores momentáneos.....	440
Implementación de pulsadores mantenidos	444
Evitando el rebote ("bounce") en los pulsadores	450
Juegos	452
Keypads digitales.....	458
USO DE LAS ENTRADAS Y SALIDAS ANALÓGICAS	461
Ejemplos con salidas analógicas	464
Control interactivo (mediante pulsadores)	466

EL MUNDO GENUINO-ARDUINO

Control interactivo (a través del canal serie)	467
Uso de LEDs RGB.....	469
Ejemplos con entradas analógicas (potenciómetros)	471
Medias y calibraciones.....	474
Entradas y salidas.....	475
Ejemplo de uso de joysticks como entradas analógicas	478
Ejemplo de uso de pulsadores como entradas analógicas	480
Cambiar el voltaje de referencia de las lecturas analógicas	484
CONTROL DE MOTORES DC	487
El chip L293	492
Módulos de control para motores DC	494
La placa TB6612FNG	495
Otros módulos	497
Shields de control para motores DC (y paso a paso)	497
El "Adafruit Motor Shield"	497
Otros shields	499
EMISIÓN DE SONIDO	502
Uso de zumbadores	502
Las funciones <i>tone()</i> y <i>noTone()</i>	505
Uso de altavoces	510
Amplificación simple del sonido	512
Sonidos pregrabados	515
La librería "SimpleSDAudio".....	515
Breve nota sobre las características de un fichero de audio.....	515
El "Wave Shield" de Adafruit.....	517
Shields que reproducen MP3.....	518
Módulos de audio	519
Reproductores de voz	523
APÉNDICES	
A. DISTRIBUIDORES DE ARDUINO Y MATERIAL ELÉCTRICO	529
Kits	532
B. CÓDIGOS IMPRIMIBLES DE LA TABLA ASCII	535
C. RECURSOS PARA SEGUIR APRENDIENDO	537
Plataforma Arduino.....	537
Electrónica general	538
Proyectos	539
ÍNDICE ANALÍTICO	541

INTRODUCCIÓN

A quién va dirigido este libro

Construir coches y helicópteros teledirigidos, fabricar diferentes tipos de robots inteligentes, crear sintetizadores de sonidos, montar una completa estación meteorológica (con sensores de temperatura, humedad, presión...), ensamblar una impresora 3D, monitorizar la eficacia de nuestro refrigerador de cervezas desde el jardín, controlar a través de Internet la puesta en marcha de la calefacción y de las luces de nuestra casa cuando estemos lejos de ella, enviar periódicamente los datos de consumo doméstico de agua a nuestra cuenta de Twitter, diseñar ropa que se ilumine ante la presencia de gas, establecer un sistema de secuencia de golpes a modo de contraseña para abrir puertas automáticamente, apagar todos los televisores cercanos de una sola vez, implementar un sistema de riego automático y autorregulado según el estado de humedad detectada en la tierra, elaborar un theremin de rayos de luz, fabricar un reloj-despertador musical, utilizar una cámara de vídeo como radar para recibir alarmas de intrusos en nuestro teléfono móvil, jugar al tres en raya mediante órdenes habladas, etc. Todo lo anterior y muchísimo más se puede conseguir con Genuino/Arduino.

Este libro está dirigido, pues, a todo aquel que quiera investigar cómo conectar el mundo físico exterior con el mundo de la electrónica y la informática, para lograr así una interacción autónoma y casi "inteligente" entre ambos mundos. Ingenieros, artistas, profesores o simples aficionados podrán conocer las posibilidades que les ofrece el ecosistema Genuino/Arduino para llevar a cabo casi cualquier proyecto que la imaginación proponga.

EL MUNDO GENUINO-ARDUINO

Este curso está pensado para usuarios con nulos conocimientos de programación y de electrónica. Se presupone que el lector tiene un nivel básico de informática doméstica (por ejemplo, sabe cómo descomprimir un archivo "zip" o cómo crear un acceso directo) pero no más. Por lo tanto, este texto es ideal para todo aquel que no haya programado nunca ni haya realizado ningún circuito eléctrico. En cierto sentido, gracias a la "excusa" de Genuino/Arduino, lo que tiene el lector en sus manos es un manual de iniciación tanto a la electrónica como a la programación básica.

El texto se ha escrito facilitando al lector autodidacta una asimilación gradual de los conceptos y procedimientos necesarios para ir avanzando poco a poco y con seguridad a lo largo de los diferentes capítulos, desde el primero hasta el último. Esta estructura hace que el texto también pueda ser utilizado perfectamente como libro de referencia para profesores que imparten cursos de Genuino/Arduino dentro de diversos ámbitos (educación secundaria, formación profesional, talleres no reglados, etc.). Aderezado con multitud de ejemplos de circuitos y códigos, su lectura permite la comprensión del universo Genuino/Arduino de una forma práctica y progresiva.

No obstante, aunque muy completo, este curso no es una referencia o compendio exhaustivo de todas las funcionalidades que ofrece el sistema Genuino/Arduino. Sería imposible abarcárlas todas en un solo volumen. El lector experimentado notará que en las páginas siguientes faltan por mencionar y explicar aspectos avanzados tan interesantes (algunos de los cuales pueden dar lugar a un libro entero por sí mismos) como el papel de Genuino/Arduino en la construcción de robots o de impresoras 3D, o las posibilidades de comunicación entre Genuino/Arduino y dispositivos con sistema Android o páginas web, por ejemplo.

Cómo leer este libro

Este curso se ha escrito teniendo en cuenta varios aspectos. Se ha procurado en la medida de lo posible escribir un manual que sea **autocontenido y progresivo**. Es decir, que no sea necesario recurrir a fuentes de información externas para comprender todo lo que se explica, sino que el propio texto sea autoexplicativo en sí mismo. Y además, que toda la información expuesta sea mostrada de forma ordenada y graduada, sin introducir conceptos o procedimientos no explicados con anterioridad. Por tanto, se recomienda una lectura secuencial, desde el primer capítulo hasta el último, sin saltos.

INTRODUCCIÓN

La metodología utilizada en este texto se basa fundamentalmente en la exposición y explicación pormenorizada de multitud de ejemplos de código **cortos y concisos**: se ha intentado evitar códigos largos y complejos, que aunque interesantes y vistosos, pueden distraer y desorientar al lector al ser demasiado inabarcables. La idea no es presentar proyectos complejos ya acabados, sino exponer de la forma más simple posible los conceptos básicos. En este sentido, se aportan multitud de enlaces para ampliar los conocimientos que no tienen espacio en el libro: muchos son los temas que se proponen (electricidad, electrónica, algoritmia, mecánica, acústica, electromagnetismo, etc.) para que el lector que tenga iniciativa pueda investigar por su cuenta.

La estructura de los capítulos es la siguiente: el primer capítulo introduce los conceptos básicos de electricidad en circuitos electrónicos, y describe –mediante ejemplos concretos– el comportamiento y la utilidad de los componentes presentes en la mayoría de estos circuitos (como pueden ser las resistencias, condensadores, transistores, placas de prototipado, etc.). El segundo capítulo expone las diferentes placas que forman el ecosistema Genuino/Arduino, los componentes que las forman y los conceptos más importantes ligados a esta plataforma. El tercer capítulo muestra el entorno de programación oficial de Genuino/Arduino y describe su instalación y configuración. El cuarto capítulo repasa la funcionalidad básica del lenguaje de programación Genuino/Arduino, proponiendo múltiples ejemplos donde se pueden observar las distintas estructuras de flujo, funciones, tipos de datos, etc., empleados por este lenguaje. El quinto capítulo muestra la diversidad de librerías oficiales que incorpora el lenguaje Genuino/Arduino, y aprovecha para profundizar en el manejo del hardware que hace uso de ellas (tarjetas SD, pantallas LCD, motores, etc.). El sexto y último capítulo se centra, finalmente, en el manejo de las entradas y salidas de una placa Genuino/Arduino, tanto analógicas como digitales, y su manipulación a través de pulsadores o potenciómetros, entre otros. Al final de este libro, por tanto, el lector tendrá todos los conocimientos necesarios para afrontar con garantías cualquier proyecto donde esté presente una placa Genuino/Arduino.

Nota aclaratoria sobre la distinción entre Genuino y Arduino

El lector ya se habrá percatado de que en los párrafos anteriores se insiste en llamar "Genuino/Arduino" al universo hasta ahora conocido simplemente como "Arduino". ¿Por qué? Por un problema legal de registro de marca. No deseamos aburrir al lector con temas tan poco atractivos como este, pero es necesario situar las cosas en el contexto adecuado para que uno pueda ubicarse convenientemente; por eso, a continuación, explicaremos brevemente la razón de la existencia del nombre "Genuino" y de su relación con el –más conocido– "Arduino".

EL MUNDO GENUINO-ARDUINO

Explicada muy brevemente, la historia es así: cuando apareció el proyecto Arduino, sus fundadores (el llamado "Arduino Team", formado inicialmente por cinco personas de las cuales hablaremos más extensamente en el capítulo 2) inicialmente se dedicaron a aspectos diferentes dentro de ese proyecto: uno se especializó más en el desarrollo del software Arduino, otro en la escritura de documentación, otro en el mantenimiento y soporte a la comunidad, otro en propuestas de nuevos diseños de placas... y uno se dedicó a fabricarlas físicamente en una planta de su propiedad bajo el amparo de la empresa italiana Smart Projects SRL. Así fue durante bastantes años, pero llegó el momento en el que el Arduino Team (excepto el fundador-fabricante) quiso externalizar la producción de las placas llegando a acuerdos con otras plantas de fabricación de terceros, las cuales estarían entonces autorizadas a poner el mismo nombre oficial "Arduino" que hasta ahora solamente podían llevar las placas fabricadas en la planta original de Smart Projects SRL. Esto dio lugar a diferentes desencuentros entre el fundador-fabricante y el resto del Arduino Team, los cuales desembocaron en una escisión en 2015.

El problema de dicha escisión es que, actualmente, ambas partes se autoproclaman el "verdadero Arduino", lo cual añade gran confusión al mercado porque actualmente hay dos páginas web y dos líneas de productos divergentes que se autodenominan "Arduino". No obstante, y ciñéndonos a la legalidad, por una cuestión de registro de nombres mercantil, el nombre "Arduino" a nivel internacional (excepto en los Estados Unidos!) está en posesión de Smart Projects SRL (actualmente rebautizada como Arduino SRL). Por tanto, ante este hecho el Arduino Team debía elegir un nuevo nombre para poder distribuir (ahora ya como una empresa legalmente llamada "Arduino LLC") sus placas por todo el mundo (excepto en Estados Unidos, donde sí puede seguir usando el nombre "Arduino"). Y ese nombre es "Genuino". Así pues, "Arduino" (entendiendo como el proyecto original del Arduino Team) y "Genuino" es lo mismo, solo que un nombre es utilizado en Estados Unidos y el otro en el resto del mundo; en el primer caso, las placas están fabricadas por la empresa Adafruit Industries y en el segundo caso dependerá de la zona del mundo donde estemos (en Asia el fabricante mayoritario es una empresa china llamada Seeed Studio, en Europa otra alemana llamada Watterott, etc.) pero esto no será relevante para nosotros.

Ya que esta lamentable situación obliga al usuario a elegir, en este libro se ha apostado por el "bando" que mantiene la misma filosofía libre del proyecto original y que está siendo apoyado por la inmensa mayoría de usuarios que ha hecho grande el universo Arduino: el proyecto Genuino. Así pues, este libro se centrará en las posibilidades que ofrecen las placas disponibles en <http://www.arduino.cc> (hogar original del Arduino Team y la comunidad Arduino, desde donde se pueden adquirir o

INTRODUCCIÓN

bien placas Arduino –si estamos en USA–, o bien las mismas placas pero rebautizadas como Genuino –si estamos fuera de USA–), y se ignorará todo aquello proveniente de <http://www.arduino.org> (hogar de los productos fabricados por Arduino SRL).

A lo largo de este libro se utilizará genéricamente la nomenclatura "Arduino" para referirnos a las placas "Genuino", aunque los productos que pueda adquirir el lector sean seguramente de este último tipo. La razón de ello es, por un lado, reivindicar con el nombre de "Arduino" la historia y los valores del proyecto original y, por otro, enfatizar un aspecto no menos importante: la mayoría de documentación actualmente existente en Internet (en forma de artículos, blogs, etc.) es anterior a la escisión y, por tanto, aún sigue utilizando el nombre "Arduino" para referirse a lo que hoy en día es Genuino.

1

ELECTRÓNICA BÁSICA

CONCEPTOS TEÓRICOS SOBRE ELECTRICIDAD

¿Qué es la electricidad?

Un electrón es una partícula subatómica que posee carga eléctrica negativa. Por lo tanto, debido a la ley física de atracción entre sí de cargas eléctricas de signo opuesto (y de repulsión entre sí de cargas eléctricas de mismo signo), cualquier electrón siempre es atraído por una carga positiva equivalente.

Una consecuencia de este hecho es que si, por razones que no estudiaremos, en un extremo (también llamado "polo") de un material conductor aparece un exceso de electrones y en el otro polo aparece una carencia de estos (equivalente a la existencia de "cargas positivas"), los electrones tenderán a desplazarse a través de ese conductor desde el polo negativo al positivo. A esta circulación de electrones por un material conductor se le llama "electricidad".

La electricidad existirá mientras no se alcance una compensación de cargas entre los dos polos del conductor. Es decir, a medida que los electrones se desplacen de un extremo a otro, el polo negativo será cada vez menos negativo y el polo positivo será cada vez menos positivo, hasta llegar el momento en el que ambos

EL MUNDO GENUINO-ARDUINO

extremos tengan una carga global neutra (es decir, estén en equilibrio). Llegados a esta situación, el movimiento de los electrones cesará. Para evitar esto, en la práctica se utiliza una fuente de alimentación externa (lo que se llama un "generador") para restablecer constantemente la diferencia inicial de cargas entre los extremos del conductor, como si fuera una "bomba". De esta manera, mientras el generador funcione, el desplazamiento de los electrones podrá continuar sin interrupción.

¿Qué es el voltaje?

En el estudio del fenómeno de la electricidad existe un concepto fundamental que es el de voltaje entre dos puntos de un circuito eléctrico (también llamado "tensión", "diferencia de potencial" o "caída de potencial"). Expliquémoslo con un ejemplo.

Si entre dos puntos de un conductor no existe diferencia de cargas eléctricas, el voltaje entre ambos puntos es cero. Si entre esos dos puntos aparece un desequilibrio de cargas (es decir, si en un punto hay un exceso de cargas negativas y en el otro una ausencia de ellas), aparecerá un voltaje entre ambos puntos, el cual será mayor a medida que la diferencia de cargas sea también mayor. Este voltaje es el responsable de la generación del flujo de electrones entre los dos puntos del conductor. No obstante, si los dos puntos tienen un desequilibrio de cargas entre sí pero están unidos mediante un material no conductor (lo que se llama un material "aislante"), existirá un voltaje entre ellos pero no habrá paso de electrones (es decir, no habrá electricidad).

Generalmente, se suele decir que el punto del circuito con mayor exceso de cargas positivas (o dicho de otra forma: con mayor carencia de cargas negativas) es el que tiene el "potencial" más elevado, y el punto con mayor exceso de cargas negativas es el que tiene el "potencial" más reducido. Pero no olvidemos nunca que el voltaje siempre se mide entre dos puntos: no tiene sentido decir "el voltaje en este punto", sino "el voltaje en este punto respecto a este otro"; de ahí sus otros nombres de "diferencia de potencial" o "caída de potencial".

Así pues, como lo que utilizaremos siempre serán las diferencias de potencial relativas entre dos puntos, el valor numérico absoluto de cada uno de ellos lo podremos asignar según nos convenga. Es decir, aunque 5, 15 y 25 son valores absolutos diferentes, la diferencia de potencial entre un punto que vale 25 y otro que vale 15, y la diferencia entre uno que vale 15 y otro que vale 5 da el mismo resultado. Por este motivo, y por comodidad y facilidad en el cálculo, al punto del circuito con potencial más reducido (el de mayor carga negativa, recordemos) se le suele dar un valor de referencia igual a 0.

CAPÍTULO 1: ELECTRÓNICA BÁSICA

También por convenio (aunque físicamente sea en realidad justo al contrario) se suele decir que la corriente eléctrica va desde el punto con potencial mayor hacia otro punto con potencial menor (es decir, que la carga acumulada en el extremo positivo es la que se desplaza hacia el extremo negativo).

Para entender mejor el concepto de voltaje podemos utilizar la analogía de la altura de un edificio: si suponemos que el punto con el potencial más pequeño es el suelo y asumimos este como el punto de referencia con valor 0, a medida que un ascensor vaya subiendo por el edificio irá adquiriendo más y más potencial respecto el suelo: cuanta más altura tenga el ascensor, más diferencia de potencial habrá entre este y el suelo. Cuando estemos hablando de una "caída de potencial", querremos decir entonces (en nuestro ejemplo) que el ascensor ha disminuido su altura respecto al suelo y por tanto tiene un voltaje menor.

La unidad de medida del voltaje es el voltio (V), pero también podemos hablar de milivoltios ($1 \text{ mV} = 0,001 \text{ V}$), o de kilovoltios ($1 \text{ kV} = 1000 \text{ V}$). Los valores típicos en proyectos de electrónica casera como los que abordaremos en este libro son de 3,3V o de 5V, aunque cuando intervienen elementos mecánicos (como motores) u otros elementos complejos, se necesitará aportar algo más de energía al circuito, por lo que los valores suelen ser algo mayores: 9V, 12V o incluso 24V. En todo caso, es importante tener en cuenta que valores más allá de 40V pueden poner en riesgo nuestra vida si no tomamos las precauciones adecuadas; en los proyectos de este libro, de todas formas, no se utilizarán nunca voltajes de esta magnitud.

¿Qué es la intensidad de corriente?

La intensidad de corriente (comúnmente llamada "corriente" a secas) es una magnitud eléctrica que se define como la cantidad de carga eléctrica que pasa en un determinado tiempo a través de un punto concreto de un material conductor. Podemos imaginar que la intensidad de corriente es similar en cierto sentido al caudal de agua que circula por una tubería: que pase más o menos cantidad de agua por la tubería en un determinado tiempo sería análogo a que pase más o menos cantidad de electrones por un cable eléctrico en ese mismo tiempo.

Su unidad de medida es el amperio (A), pero también podemos hablar de miliamperios ($1 \text{ mA} = 0,001 \text{ A}$), de microamperios ($1 \mu\text{A} = 0,001 \text{ mA}$), o incluso de nanoamperios ($1 \text{ nA} = 0,001 \mu\text{A}$).

Tal como ya hemos comentado, se suele considerar que en un circuito la corriente fluye del polo positivo (punto de mayor tensión) al polo negativo (punto de menor tensión) a través de un material conductor.

¿Qué es la corriente continua (DC) y la corriente alterna (AC)?

Hay que distinguir dos tipos fundamentales de circuitos cuando hablamos de magnitudes como el voltaje o la intensidad: los circuitos de corriente continua (o circuitos DC, del inglés "Direct Current") y los circuitos de corriente alterna (o circuitos AC, del inglés "Alternating Current").

Llamamos corriente continua a aquella en la que los electrones circulan a través del conductor siempre en el mismo sentido. Aunque comúnmente se identifica la corriente continua con la corriente constante, estrictamente solo es continua toda corriente que, tal como acabamos de decir, mantenga siempre el mismo sentido, sin importar su magnitud. Este tipo de corriente se produce en aquellos circuitos cuyos polos positivo y negativo (o dicho de otra manera, cuyos extremos de mayor y menor potencial) están conectados a un generador que los mantiene siempre en la misma polaridad.

Llamamos corriente alterna a aquella en la que los electrones cambian el sentido de su circulación de forma periódica. Este tipo de corriente se produce en aquellos circuitos cuyos extremos cambian de polaridad alternativamente a lo largo del tiempo (es decir, cuyo polo positivo se transforma en polo negativo –y viceversa– a un ritmo constante una cantidad indefinida de veces) debido a que dichos extremos están conectados a un generador (también llamado, en este caso, "alternador") responsable de estos cambios. La permuta periódica de polaridad en los polos del circuito provoca que el voltaje existente entre ellos vaya variando también de forma cíclica, adquiriendo repetidamente valores tanto positivos como negativos.

La corriente alterna es el tipo de corriente que llega a los hogares y empresas proveniente de la red eléctrica general. Esto es así porque la corriente alterna es más fácil y eficiente de transportar a lo largo de grandes distancias que la corriente continua (ya que sufre menos pérdidas de energía). Además, mediante un dispositivo llamado transformador, es posible convertir de una forma mucho más eficaz el voltaje de una corriente alterna (aumentándolo o disminuyéndolo según convenga) que si usáramos una corriente continua.

No obstante, en todos los proyectos de este libro utilizaremos tan solo corriente continua, ya que los circuitos donde podemos utilizar Arduino (y de hecho, la mayoría de circuitos electrónicos domésticos) solo funcionan correctamente con este tipo de corriente.

¿Qué es la resistencia eléctrica?

Podemos definir la resistencia eléctrica interna de un objeto cualquiera (aunque normalmente nos referiremos a algún componente electrónico que forme parte de nuestros circuitos) como su capacidad para oponerse al paso de la corriente eléctrica a través de él. Es decir, cuanto mayor sea la resistencia de ese componente, más dificultad tendrán los electrones para atravesarlo, hasta incluso el extremo de imposibilitar la existencia de electricidad.

Esta característica depende entre otros factores del material con el que está construido ese objeto, por lo que podemos encontrarnos con materiales con poca o muy poca resistencia intrínseca (los llamados "conductores", como el cobre o la plata) y materiales con bastante o mucha resistencia (los llamados "aislantes", como la madera o determinados tipos de plástico, entre otros). No obstante, hay que insistir en que aunque un material sea conductor, siempre poseerá inevitablemente una resistencia propia que evita que se transfiera el 100% de la corriente a través de él, por lo que incluso un simple cable de cobre tiene cierta resistencia interna (normalmente despreciable, eso sí) que reduce el flujo de electrones original.

La unidad de medida de la resistencia de un objeto es el ohmio (Ω), y se define como la cantidad de resistencia ofrecida por un elemento electrónico cuando, al estar sometido a un voltaje de 1V, fluye a través de él una corriente de 1A. También podemos hablar de kilohmios ($1\text{ k}\Omega = 1000\ \Omega$), de megaohmios ($1\text{ M}\Omega = 1000\text{ k}\Omega$), etc.

¿Qué es la Ley de Ohm?

La Ley de Ohm dice que si un componente eléctrico con resistencia interna, R , es atravesado por una intensidad de corriente, I , entre ambos extremos de dicho componente existirá una diferencia de potencial, V , que puede ser conocida gracias a la relación $V=I\cdot R$.

De esta fórmula es fácil deducir relaciones de proporcionalidad interesantes entre estas tres magnitudes eléctricas. Por ejemplo: se puede ver que (suponiendo que la resistencia interna del componente no cambia) cuanto mayor es la intensidad de corriente que lo atraviesa, mayor es la diferencia de potencial entre sus extremos. También se puede ver que (suponiendo en este caso que en todo momento circula la misma intensidad de corriente por el componente), cuanto mayor es su resistencia interna, mayor es la diferencia de potencial entre sus dos extremos.

EL MUNDO GENUINO-ARDUINO

Además, despejando la magnitud adecuada de la fórmula anterior, podemos obtener, a partir de dos datos conocidos cualesquiera, el tercero. Por ejemplo, si conocemos V y R, podremos encontrar I mediante $I=V/R$, y si conocemos V e I, podremos encontrar R mediante $R=V/I$.

A partir de las fórmulas anteriores debería ser fácil ver también por ejemplo que cuanto mayor es el voltaje aplicado entre los extremos de un componente (el cual suponemos que posee una resistencia de valor fijo), mayor es la intensidad de corriente que pasa por él. O que cuanto mayor es la resistencia del componente (manteniendo constante la diferencia de potencial entre sus extremos), menor es la intensidad de corriente que pasa a través de él. De hecho, en este último caso, si el valor de la resistencia es suficientemente elevado, podemos conseguir incluso que el flujo de electrones se interrumpa.

¿Qué es la potencia?

Podemos definir la potencia de un componente eléctrico/electrónico como la energía eléctrica consumida por este en un segundo; dicho consumo provocará siempre la transformación de esta energía en diferentes efectos (como calor, y/o luz, y/o sonido, y/o movimiento, etc.) dependiendo del tipo de componente que esté consumiéndola. Si estamos hablando, en cambio, de una fuente de alimentación, con la palabra potencia nos referiremos entonces a la energía eléctrica aportada por esta al circuito en un segundo.

El párrafo anterior se podría haber escrito de esta otra manera: cuando una fuente de alimentación aporta una determinada potencia (es decir, una determinada cantidad de energía por segundo), esta puede ser consumida por los distintos componentes del circuito de diversas maneras: la mayoría de veces es gastada en forma de calor debido al efecto de las resistencias internas intrínsecas de cada componente (el llamado "efecto Joule"), pero también puede ser consumida en forma de luz (si ese componente es una bombilla, por ejemplo) o en forma de movimiento (si ese componente es un motor, por ejemplo), o en forma de sonido (si ese componente es un altavoz, por ejemplo), o en una mezcla de varias.

El valor máximo posible de potencia (ya sea consumida o generada) soportado por un determinado componente o generador, respectivamente, depende del propio elemento (es decir, es intrínseco a él). Igualmente, los materiales conductores (los "cables") solo pueden consumir hasta una cantidad máxima de potencia, más allá de la cual se corre el riesgo de sobrecalentarlos y dañarlos. Siempre es buena idea, por tanto, comprobar este dato antes de conectar ningún componente, generador o cable a nuestros circuitos.

CAPÍTULO 1: ELECTRÓNICA BÁSICA

La unidad de medida de la potencia es el vatio (W), pero también podemos hablar de milivatios (1 mW= 0,001 W –orden de magnitud habitual en los proyectos donde intervienen placas Arduino–), o kilovatios (1 kW= 1000 W –orden de magnitud habitual en el ámbito del consumo eléctrico doméstico–).

Podemos calcular la potencia consumida por un componente eléctrico si sabemos el voltaje al que está sometido y la intensidad de corriente que lo atraviesa, utilizando la fórmula $P=V \cdot I$; de aquí se puede ver fácilmente que a mayor tensión y/o mayor intensidad recibida, mayor potencia consumirá el componente en cuestión. Por ejemplo, una bombilla sometida a 220V por la que circula 1A consumirá 220W (en forma de luz y calor). Por otro lado, a partir de la Ley de Ohm podemos deducir otras dos fórmulas equivalentes que nos pueden ser útiles si sabemos el valor de la resistencia R interna del componente: $P=I^2 \cdot R_{int}$ o también $P=V^2/R_{int}$.

¿Qué son las señales digitales y las señales analógicas?

Podemos clasificar las señales eléctricas (ya sean voltajes o intensidades) de varias maneras según sus características físicas. Una de las clasificaciones posibles es distinguir entre señales digitales y señales analógicas.

Señal digital es aquella que solo tiene un número finito de valores posibles (lo que se suele llamar "tener valores discretos"). Por ejemplo, si consideramos como señal el color emitido por un semáforo, es fácil ver que esta es de tipo digital porque solo puede tener tres valores concretos, diferenciados y sin posibilidad de transición progresiva entre ellos: rojo, ámbar y verde.

Un caso particular de señal digital es la señal binaria, donde el número de valores posibles solo es 2. Conocer este tipo de señales es importante porque en la electrónica es muy habitual trabajar con voltajes (o intensidades) con tan solo dos valores. En estos casos, uno de los valores del voltaje binario suele ser 0 –o un valor aproximado– para indicar precisamente la ausencia de voltaje, y el otro valor puede ser cualquiera, pero lo suficientemente distingible del 0 como para indicar sin ambigüedades la presencia de señal. De esta forma, un valor del voltaje binario siempre identifica el estado "no pasa corriente" (también llamado estado "apagado" –"off" en inglés–, BAJO –LOW en inglés–, o "0") y el otro valor siempre identifica el estado "pasa corriente" (también llamado "encendido" –"on"–, ALTO –HIGH–, o "1").

El valor de voltaje concreto que se corresponda con el estado ALTO será diferente según los dispositivos electrónicos utilizados en cada momento. Concretamente, en los proyectos de este libro solo emplearemos dispositivos

EL MUNDO GENUINO-ARDUINO

electrónicos que reconocen como estado ALTO bien el valor 3,3V, o bien el valor 5V. En el primer caso la potencia consumida es menor (y, por tanto, hay un ahorro energético bastante atractivo), así que a priori usar 3,3V como valor ALTO sería preferible a usar 5V; no obstante, en la práctica, no todos los dispositivos son capaces de funcionar reconociendo el estado ALTO a 3,3V. De hecho, es bastante habitual que en un mismo circuito no tengamos más remedio que conectar entre sí dispositivos que reconozcan el estado ALTO con niveles de voltaje diferente; en ese caso, es muy importante tener en cuenta que si sometemos un dispositivo a un voltaje demasiado elevado (por ejemplo, si aplicamos 5V como valor ALTO cuando el dispositivo solo admite 3,3V) corremos el riesgo de dañarlo irreversiblemente. Para evitar este inconveniente, podemos recurrir a "divisores de tensión" (circuitos eléctricos estudiados más adelante en este mismo capítulo) o, de una forma más sofisticada, a "convertidores de nivel" (componentes electrónicos estudiados en capítulos siguientes).

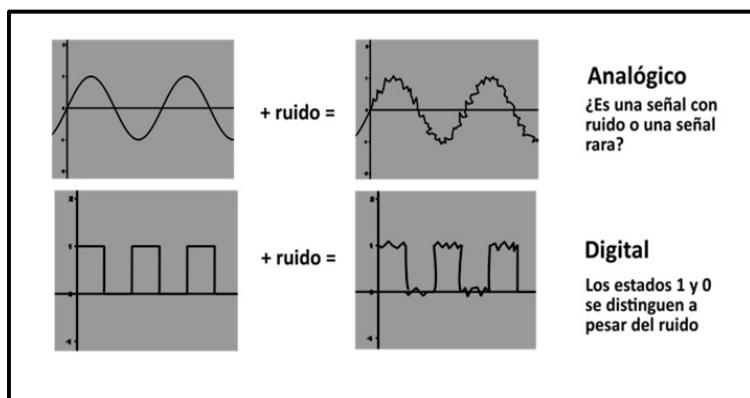
Además de los niveles ALTO y BAJO, en una señal binaria existen las transiciones entre estos niveles (de ALTO a BAJO y de BAJO a ALTO), denominadas flanco de bajada y de subida, respectivamente.

Señal analógica es aquella que tiene infinitos valores posibles dentro de un rango determinado (lo que se suele llamar "tener valores continuos"). La mayoría de magnitudes físicas (temperatura, sonido, luz...) son analógicas, así como también las más específicamente eléctricas (voltaje, intensidad, potencia...) porque todas ellas, de forma natural, pueden sufrir variaciones continuas sin saltos. De hecho, muchos de los componentes presentes en circuitos eléctricos (como por ejemplo las resistencias, los condensadores, los diodos o los transistores) son de tipo analógico.

No obstante, muchos sistemas electrónicos complejos (como un computador, por ejemplo) no tienen la capacidad de trabajar con señales analógicas: solamente pueden manejar señales digitales (especialmente de tipo binario; de ahí su gran importancia). Esto es debido a que sus componentes fundamentales (en su mayoría elementos electrónicos llamados "puertas lógicas") son de tipo binario. Así que, para que estos sistemas electrónicos puedan trabajar con señales analógicas, necesitan disponer de un elemento llamado conversor analógico-digital (comúnmente llamado "ADC") que "traduzca" (mejor dicho, "simule") las señales analógicas del mundo exterior en señales digitales entendibles por dicho sistema electrónico. También necesitarán un conversor digital-analógico (comúnmente llamado "DAC") para poder realizar el proceso inverso: transformar una señal digital interna del computador en una señal analógica y así poderla emitir al mundo físico. Un ejemplo del primer caso sería la grabación de un sonido mediante un micrófono, y uno del segundo caso sería la reproducción de un sonido pregrabado mediante un altavoz.

Sobre los métodos utilizados para realizar estas conversiones de señal analógica a digital, y viceversa, ya hablaremos extensamente más adelante, pero lo que debemos saber ya es que, sea cual sea el método utilizado, siempre existirá una pérdida de información (de "calidad") durante el proceso de conversión de la señal. Esta pérdida aparece porque es matemáticamente imposible realizar una transformación perfecta de un número infinito de valores (señal analógica) a un número finito (señal digital) debido a que, por fuerza, varios valores de la señal analógica deben "colapsar" en un único valor indistinguible de la señal digital.

A pesar de lo anterior, la razón por la cual la mayoría de sistemas electrónicos utilizan para funcionar señales digitales en vez de señales analógicas es porque, además de ser así más sencillos de diseñar y construir, las señales digitales tienen una gran ventaja respecto a las señales analógicas: son más inmunes al ruido. Por "ruido" se entiende cualquier variación no deseada de la señal, y es un fenómeno que ocurre constantemente debido a una gran multitud de factores. El ruido modifica la información que aporta una señal y afecta en gran medida al correcto funcionamiento y rendimiento de los dispositivos electrónicos. Si la señal es analógica, el ruido es mucho más difícil de tratar y la recuperación de la información original se complica.



¿Qué son las señales periódicas y las señales aperiódicas?

Otra clasificación que podemos hacer con las señales eléctricas es dividirlas entre señales periódicas y aperiódicas. Llamamos señal periódica a aquella que se repite tras un cierto periodo de tiempo (T) y señal aperiódica a aquella que no se repite. En el caso de las primeras (las más interesantes con diferencia), dependiendo de cómo varíe la señal a lo largo del tiempo, esta puede tener una "forma" concreta (senoidal –es decir, que sigue el dibujo de la función seno–, cuadrada, triangular, etc.).

EL MUNDO GENUINO-ARDUINO

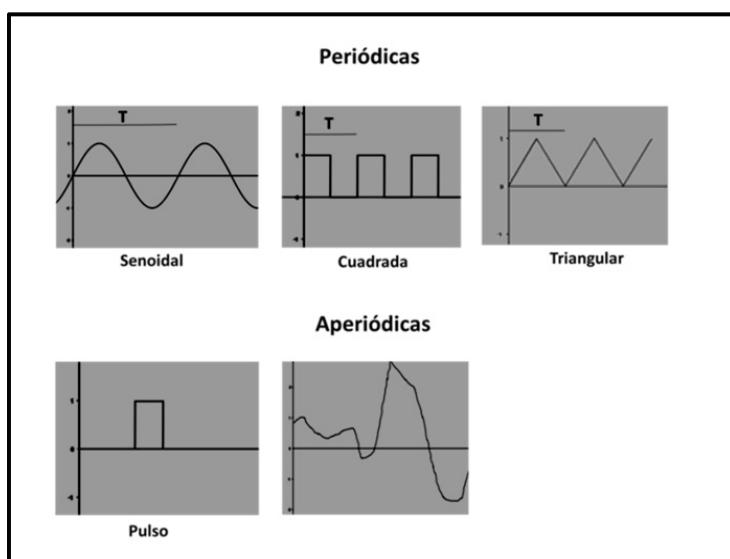
Las señales periódicas (tales como el voltaje AC recibido por las tomas de corriente en hogares y empresas, o las ondas sonoras, por poner dos ejemplos) tienen una serie de características que debemos identificar y definir para poder trabajar con ellas de una forma sencilla:

Frecuencia (f): es el número de veces que la señal se repite en un segundo. Se mide en hercios (Hz), o sus múltiplos (como kilohercios o megahercios). Por ejemplo, si decimos que una señal es de diez hercios, significa que se repite diez veces cada segundo.

Período (T): es el tiempo que dura un ciclo completo de la señal, antes de repetirse otra vez. Es el inverso de la frecuencia ($T = 1/f$) y se mide en segundos.

Valor instantáneo: es el valor concreto que toma la señal (voltaje, intensidad, etc.) en cada instante.

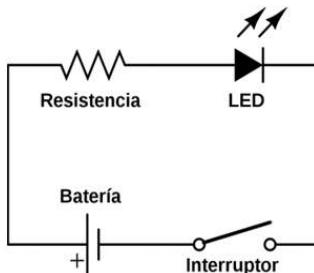
Valor medio: es un valor calculado matemáticamente realizando la media de los diferentes valores que ha ido teniendo la señal a lo largo de un tiempo concreto. Algunos componentes electrónicos (por ejemplo, algunos motores) responden no al valor instantáneo sino al valor medio de la señal.



CIRCUITOS ELÉCTRICOS BÁSICOS

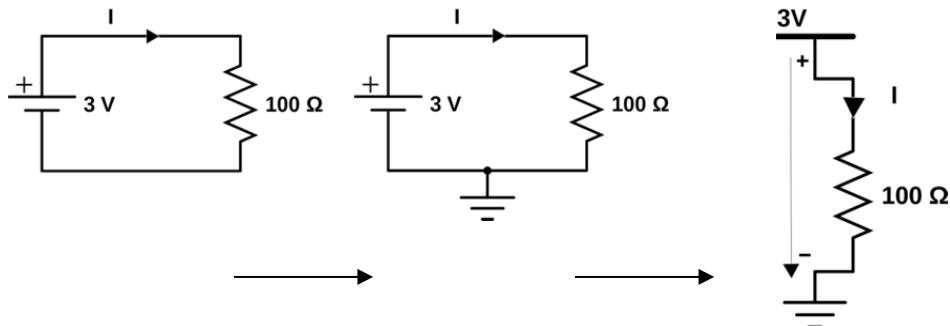
Representación gráfica de circuitos

Para describir de una forma sencilla y clara la estructura y la composición de un circuito eléctrico se utilizan esquemas gráficos. En ellos se representa cada dispositivo del circuito mediante un símbolo estandarizado y se dibujan todas las interconexiones existentes entre ellos. Por ejemplo, un circuito muy simple sería:



En el esquema anterior podemos apreciar cuatro dispositivos (presentes prácticamente en cualquier circuito) representados por su símbolo convencional: una pila o batería (cuya tarea es alimentar eléctricamente al resto de componentes), una resistencia (componente específicamente diseñado para oponerse al paso de la corriente, de ahí su nombre), un LED (componente que se ilumina cuando recibe corriente) y un interruptor. En este ejemplo, la batería creará la diferencia de potencial necesaria entre sus dos extremos –también llamados "bornes" o "polos”– para que se genere una corriente eléctrica, la cual surgirá desde su polo positivo (el marcado con el signo "+"), pasará a través de la resistencia, pasará seguidamente a través del LED (iluminándolo, por tanto) y llegará a su destino final (el polo negativo de la batería) siempre y cuando el interruptor cierre el circuito.

Por otro lado, los circuitos se pueden representar alternativamente de una forma ligeramente diferente a la mostrada anteriormente, utilizando para ello el concepto de "tierra" (también llamado "masa"). La "tierra" ("ground" en inglés) es simplemente un punto del circuito que elegimos arbitrariamente como referencia para medir la diferencia de potencial existente entre este y cualquier otro punto del circuito. En otras palabras: el punto donde diremos que el voltaje es 0. Por utilidad práctica, normalmente el punto de tierra se asocia al polo negativo de la pila. Este nuevo concepto nos simplificará muchas veces el dibujo de nuestros circuitos, ya que si representamos el punto de tierra con el símbolo \equiv , los circuitos se podrán dibujar de la siguiente manera (por simplicidad hemos suprimido el interruptor y el LED del esquema anterior):



También podremos encontrarnos con esquemas eléctricos que muestren intersecciones de cables. En este caso, deberemos fijarnos si aparece dibujado un círculo en el punto central de la intersección. Si es así, se nos estará indicando que los cables están física y eléctricamente conectados entre sí. Si no aparece dibujado ningún círculo en el punto central de la intersección, se nos estará indicando que los cables son vías independientes que simplemente se cruzan en el espacio.

Circuitos abiertos, cerrados y cortocircuitos

Aclaremos ahora lo que significa "cerrar un circuito". Ya sabemos que si existe una diferencia de potencial, aparecerá una corriente eléctrica que siempre circula desde el polo positivo de la pila hasta el negativo. Pero esto solo es posible si existe entre ambos polos un camino (el circuito propiamente dicho) que permita el paso de dicha corriente. Si el circuito está abierto, a pesar de que la batería esté funcionando, la corriente no fluirá. La función de los interruptores es precisamente cerrar o abrir el circuito para que pueda pasar la corriente o no, respectivamente. En el esquema siguiente esto se ve más claro:



Es importante tener en cuenta que la presencia de la resistencia R en el circuito del esquema anterior es imprescindible: si construyéramos el mismo circuito sin ella (es decir, solamente usando la batería y el interruptor), al cerrar este provocaríamos lo que se llama un "cortocircuito", con consecuencias muy graves

(como el sobrecalentamiento, el derretimiento o incluso la explosión de alguna parte del circuito). La razón de esto es la Ley de Ohm: al cumplirse que $I=V/R$, si un circuito no ofrece resistencia ($R \approx 0$), la intensidad de la corriente que circula por él tiende a infinito ($I \approx V/0$), provocando el mencionado cortocircuito.

El papel que cumple la resistencia R en el circuito del esquema anterior es el de "carga": todo circuito cerrado ha de tener una "carga" cuya función es ofrecer una mínima resistencia para evitar un cortocircuito. No es necesario que la "carga" sea necesariamente una resistencia "tal cual": puede ser cualquier componente electrónico (o cualquier combinación de ellos) porque todos incluyen internamente un cierto grado de resistencia intrínseca. De hecho, por "carga" se suele entender toda la parte del circuito "que funciona" gracias a la fuente de alimentación y que se comporta como una gigantesca resistencia (que realiza una tarea útil).

Conexiones en serie y en paralelo

Los distintos dispositivos presentes en un circuito pueden conectarse entre sí de varias formas. Las más básicas son la "conexión en serie" y la "conexión en paralelo". De hecho, cualquier otro tipo de conexión, por compleja que sea, es una combinación de alguna de estas dos.

Dos o más componentes están conectados en paralelo si sus respectivos extremos por donde "entra" la corriente están conectados entre ellos directamente (y lo mismo ocurre con sendos extremos por donde "sale" la corriente). En este caso:

1. El flujo total de electrones se distribuirá a través de los diferentes componentes (normalmente, de forma desigual). Por tanto, la intensidad de corriente total será la suma de las intensidades que pasan por cada componente (hecho que se conoce –en otros términos– como "1^a Ley de Kirchhoff").
2. Entre los extremos de cualquiera de los componentes existirá la misma diferencia de potencial.

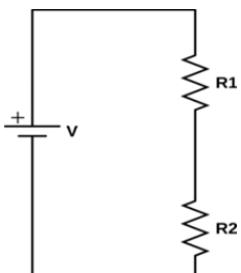
Dos o más componentes están conectados en serie si el extremo por donde "sale" la corriente de uno de ellos está conectado directamente al extremo por donde "entra" al siguiente componente. En este caso:

1. La intensidad de corriente que circulará por todos los componentes será siempre la misma (ya que solo existe un camino posible para el flujo de los electrones).

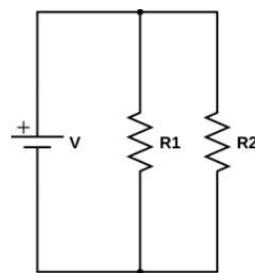
EL MUNDO GENUINO-ARDUINO

2. La diferencia de potencial total existente se repartirá entre los diferentes componentes (normalmente, de forma desigual), de manera que cada uno trabaje sometido a una parte de esa tensión total. Por tanto, la suma de las tensiones en cada componente dará como resultado dicha tensión total (hecho que se conoce –en otros términos– como "2^a Ley de Kirchhoff").

Se puede entender mejor la diferencia mediante los siguientes esquemas, en los que se puede ver la conexión en serie y en paralelo de dos resistencias:

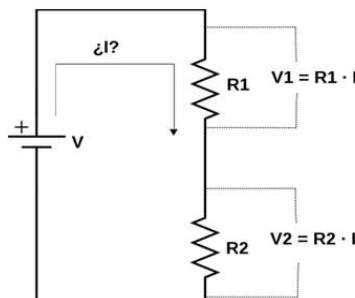


Conexión en serie



Conexión en paralelo

Aplicando las Leyes de Kirchhoff y la Ley de Ohm podemos obtener el valor de alguna magnitud eléctrica (V , I o R) si conocemos previamente el valor de alguna otra involucrada en el mismo circuito. Veamos esto usando como ejemplo el siguiente circuito, donde aparecen dos resistencias en serie:

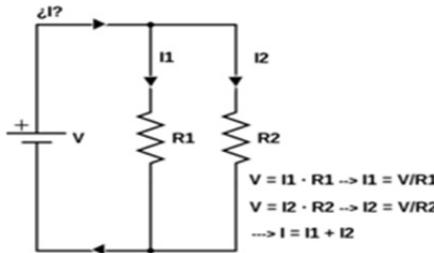


$$V = V_1 + V_2 = (R_1 + R_2) \cdot I$$

En el esquema anterior V_1 representa la caída de tensión entre los extremos de R_1 y V_2 la caída de tensión entre los extremos de R_2 . Si tenemos por ejemplo una fuente de alimentación eléctrica (una pila) que aporta un voltaje de 10V y dos resistencias cuyos valores son $R_1=1\Omega$ y $R_2=4\Omega$, respectivamente, para calcular la intensidad que circula tanto por R_1 como por R_2 (recordemos que es la misma porque

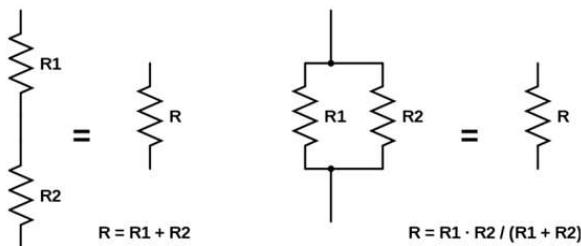
solo existe un único camino posible) simplemente deberemos realizar la siguiente operación: $I = 10V/(1\Omega+4\Omega) = 2A$, tal como se muestra en el esquema anterior.

Veamos ahora otro circuito, donde aparecen también dos resistencias, pero esta vez conectadas en paralelo:



En el esquema anterior I_1 representa la intensidad de corriente que atraviesa R_1 e I_2 la intensidad de corriente que atraviesa R_2 . Si tenemos por ejemplo una fuente de alimentación eléctrica (una pila) que aporta un voltaje de 10V y dos resistencias cuyos valores son $R_1=1\Omega$ y $R_2=4\Omega$, respectivamente, para calcular la intensidad que circula por R_1 deberíamos realizar (tal como se muestra en el esquema) la siguiente operación: $I_1 = 10V/1\Omega = 10A$; para calcular la intensidad que circula por R_2 deberíamos hacer: $I_2 = 10V/4\Omega = 2,5A$; y la intensidad total que circula por el circuito sería la suma de las dos: $I = I_1 + I_2 = 10A + 2,5A = 12,5A$.

A partir de los ejemplos anteriores, podemos deducir un par de fórmulas que nos vendrán bien a lo largo de todo el libro para simplificar los circuitos. Si tenemos dos (o más) resistencias (R_1 , R_2 , R_3 ,...) conectadas en serie o en paralelo, es posible sustituirlas en nuestros cálculos por una sola resistencia cuyo comportamiento sea totalmente equivalente. En el caso de la conexión en serie, el valor de dicha resistencia (R) es simplemente $R=R_1+R_2+R_3+\dots$, y en el caso de la conexión en paralelo, su valor equivalente se calcularía mediante la fórmula $1/R=1/R_1+1/R_2+1/R_3+\dots$ (la cual se puede simplificar, en el caso de haber solamente dos resistencias, en la expresión $R=R_1 \cdot R_2 / (R_1+R_2)$), tal como se puede ver en el siguiente diagrama).



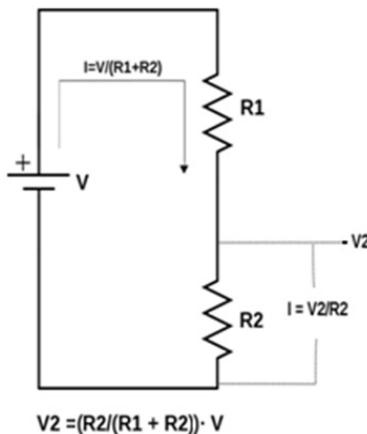
EL MUNDO GENUINO-ARDUINO

Un dato interesante de tener en cuenta (que se deduce de la propia fórmula) es que cuando se conectan resistencias en paralelo, el valor de R resultante siempre es menor que el menor valor de las resistencias implicadas.

El divisor de tensión

El "divisor de tensión" no es más que un circuito formado por una resistencia conectada en serie con cualquier otro dispositivo eléctrico. Su intención es reducir la caída de tensión a la que se ve sometido dicho dispositivo, estableciéndola en un valor seguro para no dañarlo. Dicho de otra forma: el "divisor de tensión" sirve para obtener un voltaje menor que un cierto voltaje original.

La mayor o menor cantidad de reducción que consigamos en el voltaje final dependerá del valor de la resistencia que utilicemos como divisor: a mayor valor de resistencia, mayor reducción. De todas formas, hay que tener en cuenta además que el voltaje obtenido asimismo depende del valor del voltaje original: si aumentamos este, aumentaremos proporcionalmente aquel también. Todos estos valores los podemos calcular fácilmente usando un ejemplo concreto, como el del esquema siguiente:



Tal como se puede ver, tenemos una fuente de alimentación eléctrica (una pila) que aporta un voltaje de 10V y dos resistencias cuyos valores son $R_1=1\Omega$ (la cual hará de divisor de tensión) y $R_2=4\Omega$, respectivamente. Sabemos además que la intensidad I es siempre la misma en todos los puntos del circuito –ya que no hay ramificaciones en paralelo–. Por lo tanto, para calcular V_2 (es decir, el voltaje aplicado sobre R_2 , el cual ha sido rebajado respecto al aportado por la pila gracias a R_1), nos podemos dar cuenta de que $I=V_2/R_2$ y que $I=V/(R_1+R_2)$, por lo que de aquí es fácil obtener que $V_2=(R_2/V)/(R_1+R_2)$. Queda entonces claro de la expresión anterior lo dicho en el párrafo anterior: que V_2 siempre será proporcionalmente menor a V , y según sea

R_1 mayor, V_2 será menor (de hecho, si R_1 tuviera un valor suficientemente grande, V_2 tendería a 0). Otra consecuencia interesante de la expresión anterior es que si, por el contrario, R_1 tuviera un valor mucho más pequeño que el de R_2 , no se produciría reducción en la tensión, ya que en ese caso $V_2 \approx V$.

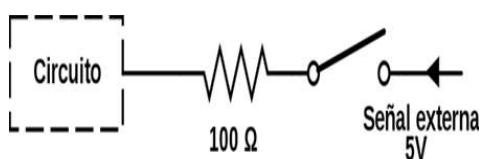
Las resistencias "pull-up" y "pull-down"

Muchas veces, los circuitos eléctricos, además de recibir la señal de alimentación de una determinada fuente, reciben otras señales eléctricas del exterior a través de "entradas" diseñadas para ello. Estas señales externas suelen ser de tipo binario (es decir, suelen tener dos posibles valores: ALTO o BAJO) y pueden servir para multitud de cosas: para activar o desactivar partes del circuito, para transmitir al circuito determinada información de su entorno, etc. El accionamiento de un interruptor, por ejemplo, es uno de los casos típicos donde el valor de la señal externa recibida por la entrada del circuito puede variar (o ALTO o BAJO) según si dicho interruptor está cerrado o no.

Una resistencia "pull-up" (o "pull-down") es una resistencia normal, solo que lleva ese nombre por la función que cumple: sirve para asumir un valor estable de la señal recibida en una entrada del circuito cuando por ella no se detecta ningún valor concreto definido (ni ALTO ni BAJO), que es lo que ocurre cuando la entrada no está conectada a nada (es decir, está "al aire"). Así pues, este tipo de resistencias aseguran que los valores binarios recibidos no fluctúan sin sentido en ausencia de señal de entrada.

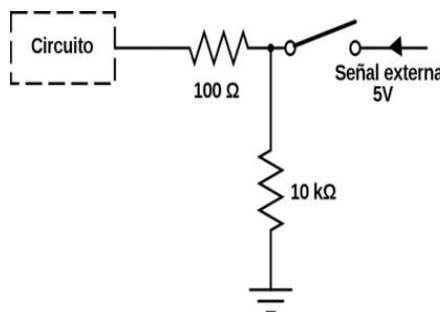
La diferencia entre una resistencia "pull-up" y una "pull-down" solamente está en su ubicación dentro del circuito (ya que ambas persiguen el mismo objetivo): una resistencia "pull-up" se conecta directamente al origen de la señal externa y una resistencia "pull-down" se conecta directamente a tierra (ver diagramas siguientes). La elección de una u otra dependerá de las circunstancias particulares de nuestro montaje.

Veamos un ejemplo concreto de la utilidad de una resistencia "pull-down". Supongamos que tenemos un circuito como el siguiente (donde la resistencia de 100 ohmios no es más que un divisor de tensión colocado en la entrada del circuito para protegerla):



EL MUNDO GENUINO-ARDUINO

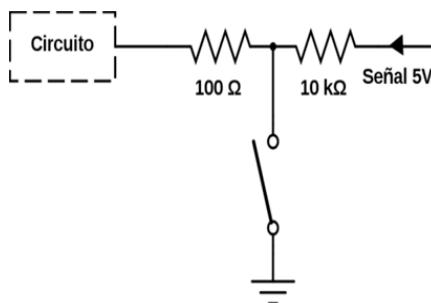
Cuando el interruptor está pulsado, la entrada del circuito anterior estará conectada a una señal válida en su estado ALTO (correspondiente en este ejemplo a 5V). En cambio, si el interruptor se deja de pulsar, el circuito se abrirá y la entrada del circuito no estará conectada a nada. Esto implica que habrá una señal de entrada fluctuante (también llamada "flotante" o "inestable") que no nos interesa. Una solución posible sería colocar una resistencia "pull-down" (cuyo valor en este ejemplo será de 10 KΩ) así:



De esta manera, cuando el interruptor esté pulsado, la entrada del circuito seguirá estando conectada a la misma señal ALTO válida de antes pero lograremos además que cuando el interruptor se deje de pulsar, la entrada del circuito esté conectada a la resistencia "pull-down", la cual tira hacia tierra (que es una referencia BAJO siempre fija de 0V).

Alguien podría pensar que cuando el interruptor esté pulsado, el circuito recibirá la señal de entrada pero también estará conectado a tierra a través de la resistencia "pull-down"... ¿Qué pasa realmente entonces? Pues que la señal externa (que no deja de ser una corriente de electrones) se encontrará con una oposición ejercida por la resistencia "pull-down" (siempre que su valor sea suficientemente grande) provocando que esta se desvíe siempre hacia la entrada del circuito. De hecho, si hubiéramos conectado la entrada del circuito a tierra directamente sin usar la resistencia "pull-down", la señal externa se habría dirigido a tierra sin pasar por la entrada del circuito porque por ese camino habría encontrado nula resistencia (provocando, por otro lado, la aparición de un cortocircuito).

Con una resistencia "pull-up" se puede conseguir lo mismo, tal como muestra el siguiente esquema. En este caso, al pulsar el interruptor, la señal exterior fluirá a través de él porque por allí encontrará un camino más directo hacia tierra (haciendo, por tanto, que la entrada del circuito reciba un valor BAJO estable), mientras que si dejamos abierto el interruptor, la entrada del circuito recibirá directamente la señal ALTO del exterior.



En los ejemplos anteriores hemos utilizado resistencias "pull-up" o "pull-down" de 10KΩ. Aunque es una norma bastante habitual utilizar este valor concreto en proyectos de electrónica donde se trabaja en el rango de los 5V, es posible emplear otros valores diferentes siempre y cuando (pondremos el ejemplo concreto de una resistencia "pull-up") cumplan dos condiciones: que sean lo suficientemente grandes para que cuando se pulse el interruptor, este no sea atravesado por demasiada corriente y lo suficientemente pequeños para que, cuando se deje sin pulsar, la entrada del circuito reciba una tensión que siga siendo reconocible como ALTO.

FUENTES DE ALIMENTACIÓN ELÉCTRICA

Llamamos fuente de alimentación eléctrica (o "generador") al elemento responsable de crear la diferencia de potencial necesaria para que fluya la corriente eléctrica por un circuito y así puedan funcionar los dispositivos conectados a este. Ya se comentó anteriormente que la placa Arduino solo funciona con corriente continua, así que para nuestros proyectos necesitaremos fuentes de alimentación que generen este tipo de corriente. De entre todas las posibles, las más habituales (y las que estudiaremos a continuación) son dos: las pilas o baterías y los adaptadores AC/DC.

Tipos de pilas/baterías

El término "pila" sirve para denominar a los generadores de electricidad basados en procesos químicos normalmente no reversibles; por tanto, son generadores no recargables. Por el contrario, el término "batería" se aplica generalmente a dispositivos electroquímicos semi-reversibles que permiten ser recargados. No obstante, estos términos no son una definición formal estricta. El término "acumulador" se aplica indistintamente a uno u otro tipo (así como a otros tipos de generadores de tensión, como los condensadores eléctricos) siendo pues un término neutro capaz de englobar y describir a todos ellos.

EL MUNDO GENUINO-ARDUINO

Si distinguimos las pilas/baterías por la disolución química interna responsable de la generación de la diferencia de potencial entre sus polos, encontraremos que las pilas ("acumuladores no recargables") más extendidas actualmente en el mercado son las de tipo alcalino, y las baterías ("acumuladores recargables") más habituales son por un lado las de níquel-cadmio (NiCd) y las –mucho más recomendables– de níquel-hidruro metálico (NiMH), y por otro las de ion-litio (Li-ion) y las de polímero de ion-litio (LiPo). De todos estos tipos de baterías, las LiPo son las que, con diferencia, tienen una densidad de carga más elevada (es decir, son las que, a igual peso, contienen más carga y, por tanto, ofrecen más autonomía) pero son las más caras.

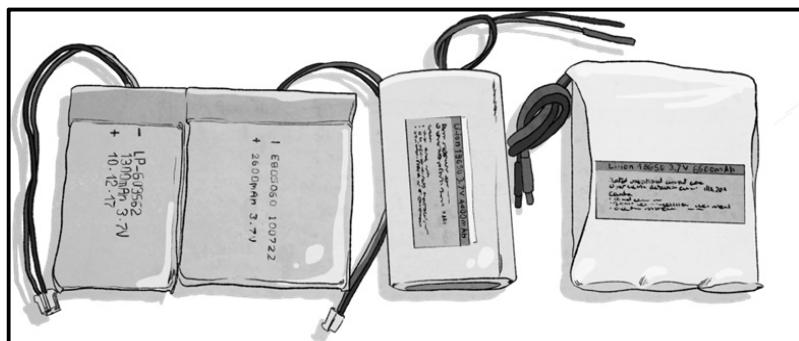
La industria internacional sigue unas normas comunes de estandarización para la fabricación de pilas de tipo alcalino y baterías de tipo NiCd/NiMH que definen unos determinados tamaños, formas y voltajes preestablecidos, de manera que se puedan utilizar sin problemas en cualquier aparato eléctrico a nivel mundial. En este sentido, los tipos de pilas más habituales son las de tipo D (LR20), C (LR14), AA (LR06) y AAA (LR03), todas ellas generadoras de 1,5V (en el caso de ser alcalinas) o 1,2V (en el caso de ser de tipo NiCd/NiMH) y todas ellas de forma cilíndrica (aunque de dimensiones diferentes; de hecho, se han listado de mayor tamaño a menor). También son frecuentes las de tipo PP3 (6LR61), que generan 9V y tienen forma de prisma rectangular; y las de tipo 3R12 (de "petaca") que generan 4,5V y tienen forma cilíndrica achatada. En la imagen siguiente se pueden apreciar, de izquierda a derecha, acumuladores –alcalinos– de tipo D, C, AA, AAA, AAAA y PP3:



Por otro lado, las baterías de tipo LiPo y Li-ion son comercializadas en una gran variedad de formas y tamaños. Podemos encontrar, por ejemplo, baterías Li-ion en forma de cilindros duros (similares en aspecto al formato AA, aunque con otros tamaños... Uno de los más populares es el formato 18650), en forma de prismas duros rectangulares (comunes sobre todo en computadores portátiles), en formato

CAPÍTULO 1: ELECTRÓNICA BÁSICA

PP3, etcétera, etcétera. En este libro, no obstante, solo manejaremos baterías Li-ion con el aspecto de carcásas redondeadas y achatadas con dos cables sobresalientes haciendo de bornes positivo (generalmente de color rojo) y negativo (generalmente de color negro) y baterías LiPo con el aspecto (muy común, por otro lado) de delgados rectángulos dentro de una bolsa plateada con, también, dos cables sobresalientes haciendo de bornes positivo (rojo) y negativo (negro). En ambos casos, estos dos cables sobresalientes suelen terminar unidos en un único conector macho de 2 pines llamado genéricamente "JST de 2mm" (o, más técnicamente, "JST-PH") que sirve para facilitar el acoplamiento de la batería al circuito a alimentar (o al circuito recargador) gracias a la presencia en este de un conector hembra del mismo tipo; así, los conectores JST permiten un empalme duradero, compacto y difícil de realizar en sentido contrario. Por otro lado, cada unidad elemental de batería LiPo o Li-ion ofrece un voltaje de 3,7V, por lo que en el mercado es habitual encontrar baterías de dicho valor nominal o de un múltiple de este (7,4V, 11,1V, etc.). Las baterías LiPo son más ligeras que las Li-ion pero suelen tener una capacidad menor; por eso las primeras se suelen utilizar en aparatos pequeños como teléfonos móviles y las segundas en cargadores de computadores portátiles y similares. En la imagen siguiente, a la izquierda se muestran dos baterías LiPo y a la derecha dos baterías cilíndricas Li-ion:



También hemos de indicar la existencia de las pilas/baterías de tipo "botón". De entre las primeras (es decir, los acumuladores no recargables) podemos destacar las fabricadas con litio-dióxido de manganeso (cuya nomenclatura empieza con "CR" –CR2032, CR2477, etc. – y, cuyo voltaje generado –aunque cada modelo tenga una capacidad y un tamaño diferente– siempre es de 3V), las fabricadas con óxido de plata (cuya nomenclatura comúnmente empieza con "SR" o "SG" –SR44, SR58... según sus dimensiones– y cuyo voltaje generado es de 1,5V) y las de tipo alcalinas (cuyo código comúnmente empieza por "LR" o "AG" y cuyo voltaje generado es también de 1,5V). Por otro lado, como baterías recargables de tipo "botón" podemos destacar las fabricadas con Li-ion (cuya nomenclatura empieza con "LIR" y cuyo voltaje generado

EL MUNDO GENUINO-ARDUINO

es de 3,6V). En cualquier caso, sea del tipo que sea, en todas las pilas/baterías botón el terminal negativo es la tapa y el terminal positivo es el metal de la otra cara, el cual generalmente está identificado con un signo "+".

Existen muchos más tipos de pilas/baterías que no trataremos en este libro por no ser relevantes para nuestros proyectos. No obstante, si el lector deseara conocerlos, puede consultar los artículos de la Wikipedia "List_of_battery_types" y "List_of_battery_sizes", además de la estupenda web <http://batteryuniversity.com> (donde podrá profundizar también en muchos detalles técnicos específicos de cada tipo de pila/batería).

Voltaje de corte, capacidad y capacidad de las pilas/baterías

Hay que tener en cuenta que el voltaje que aportan las distintas pilas es un valor "nominal": es decir, por ejemplo una pila AA de 1,5V en realidad al principio de su vida útil genera unos 1,6V, rápidamente desciende a 1,5V y entonces poco a poco va descendiendo hasta 1V, momento en el cual ha alcanzado el mínimo valor aceptable de voltaje suministrado (el llamado voltaje de corte o "cut-off"); si una pila/batería no es capaz de generar una tensión mayor que el voltaje de corte podemos considerarla "gastada" y, por tanto, conviene sustituirla para evitar problemas. Esto pasa con todos los tipos de batería: por ejemplo, una batería LiPo marcada como "3,7V/(4,2V)" indica que inicialmente es capaz de aportar un voltaje máximo de 4,2V pero rápidamente desciende a 3,7V, el cual será su voltaje medio durante la mayor parte de su vida útil, hasta que finalmente baje rápidamente hasta los 3V (su voltaje "cut-off") y automáticamente deje de funcionar. En este sentido, es útil consultar la documentación oficial ofrecida por el fabricante para cada batería particular (el llamado "datasheet" de la batería) para saber la variación del voltaje aportado en función del tiempo de funcionamiento y su valor concreto de voltaje "cut-off".

Además de la tensión generada por una pila/batería (que asumiremos a partir de ahora, por simplificar, siempre constante) hay que conocer otra característica intrínseca de ellas muy importante: la cantidad de carga eléctrica (comúnmente llamada "capacidad") que una pila/batería es capaz de almacenar. Este valor se mide en amperios-hora (Ah), o miliamperios-hora (mAh) y nos permite saber aproximadamente cuánta intensidad de corriente puede aportar ininterrumpidamente una pila/batería a un circuito durante el periodo de tiempo que transcurre desde que está completamente cargada hasta que se descarga del todo (o dicho matemáticamente: *capacidad batería = intensidad aportada x tiempo de descarga*). En este sentido, hay que tener en cuenta que, aunque la capacidad de una

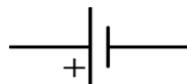
pila/batería depende de su propia constitución interna, la intensidad aportada a un circuito variará según la necesidad de consumo eléctrico que demande a cada momento el conjunto de componentes que forman ese circuito (en otras palabras: *intensidad aportada = consumo circuito*). Por tanto, de la expresión anterior podemos deducir que *tiempo de descarga = capacidad batería / consumo circuito*. Así pues, según la definición de capacidad, una pila/batería con una capacidad de 1Ah podrá ofrecer en teoría una intensidad constante de 1A durante una hora, pero si el consumo total del circuito solamente fuera de 0,1A –este dato lo podemos conocer gracias a un multímetro–, entonces, de la expresión matemática anterior se deduce que esa misma pila/batería podría estar funcionando durante 10 horas. Igualmente, si el consumo del circuito fuera de 0,01A, esa misma pila/batería podría aportar esa intensidad durante 100 horas, etc., etc.; eso sí, el voltaje generado en todos estos casos sería siempre el mismo (el valor nominal de la pila/batería).

Desgraciadamente, la capacidad de una pila/batería es un valor solamente orientativo porque cuanta más intensidad aporte una pila/batería, su tiempo de funcionamiento se reducirá en una proporción mucho mayor a la marcada por su capacidad. Por ejemplo, una pila botón de 1Ah es incapaz de aportar 1A durante una hora entera (ni tan siquiera 0,1A en 10 horas) porque se agota mucho antes, pero en cambio, no tiene problemas en aportar 0,001A durante 1000 horas. Para saber la intensidad máxima de corriente aportada por una pila/batería que aún respeta su valor nominal de Ah, deberemos consultar la documentación del fabricante (el "datasheet" de la batería). Esta intensidad de corriente "umbral" se suele llamar "capacidad", y viene expresada en unidades C, donde una unidad C se corresponde con el valor de Ah de esa batería dividido entre una hora. Así, la unidad C de una batería con carga de 2Ah será de 2A, siendo su capacidad concreta una cantidad determinada de unidades C, consultable en el datasheet (1C, 2C...). Si tenemos entonces, por ejemplo, una batería de 2Ah y 0,5C y otra de 2Ah y 2C, la primera podrá aportar sin problemas una corriente estable de 1A (su valor de capacidad) durante 2 horas y la segunda una de 4A (su valor de capacidad) durante 30 minutos, pero si aumentaran la intensidad aportada más allá de sus capacidades respectivas, los correspondientes tiempos de descarga serían proporcionalmente mucho menores. Sabido esto, hay que tener en cuenta por ejemplo que las pilas botón tienen una capacidad muy pequeña (0,01C es un valor habitual), por lo que si son forzadas a aportar mucha intensidad en un momento dado, su vida se reducirá drásticamente.

En cualquier caso, aun teniendo en cuenta el valor de la capacidad de una pila/batería solamente dentro de su capacidad, este valor debe ser tomado solo como una aproximación.

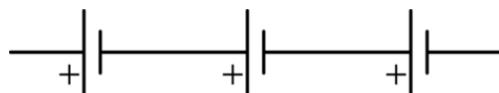
Conexiones de varias pilas/baterías

Ya hemos visto en diagramas anteriores que el símbolo que se suele utilizar en el diseño de circuitos electrónicos para representar una pila o batería es:

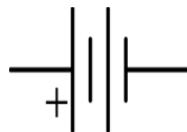


donde la parte más larga (y a veces pintada más gruesa) del dibujo representa el polo positivo de la fuente. A menudo se omite el símbolo "+".

Cuando hablamos de conectar pilas "en serie" queremos decir que conectamos el polo negativo de una con el polo positivo de otra, y así, de tal forma que finalmente tengamos un polo positivo global por un lado y un polo negativo global por otro. En esta figura se puede entender mejor:

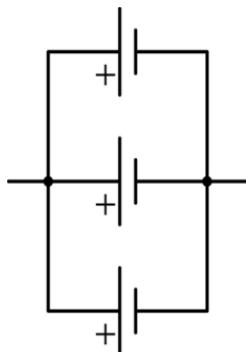


El voltaje total aportado por baterías conectadas en serie es la suma de sus voltajes individuales. Por tanto, la conexión en serie de baterías es útil cuando necesitamos tener una batería que genere un determinado voltaje relativamente elevado (por ejemplo, 12V) y solo disponemos de pilas de menor voltaje (por ejemplo, de 1,5V). Así pues, para obtener (pongamos por caso) 12V a partir de pilas de 1,5V, necesitaríamos 8 unidades ($1,5V \cdot 8 = 12V$). De hecho, las pilas comerciales de 4,5V y 9V (y de 6V y 12V, que también las hay) suelen fabricarse conectando internamente en serie pilas de 1,5V. Por eso, muchas veces veremos el siguiente símbolo (en vez del anteriormente mostrado) representando una pila:



No obstante, también hay que tener en cuenta que la capacidad total (es decir, los mAh del conjunto de pilas en serie) no aumenta: seguirá siendo exactamente la misma que la que tenga una batería de ese conjunto de forma individual e independiente. Este hecho es importante porque, en general, los circuitos que necesitan ser alimentados con grandes voltajes tienen un mayor consumo eléctrico, por lo que (en virtud de la fórmula del apartado anterior) el tiempo de funcionamiento de una fuente formada por pilas en serie será bastante reducido.

Otra manera de conectar entre sí diferentes pilas individuales es en paralelo: en esta configuración, todos los polos del mismo signo están unidos entre sí. Es decir: por un lado se conectan los polos negativos de cada pila y por otro lado se conectan todos los polos positivos, siendo estos dos puntos comunes de unión los polos negativo y positivo globales:



Un conjunto de pilas en paralelo ofrece el mismo voltaje que una sola pila individual (es decir, si tenemos por ejemplo cuatro pilas de 1,5V conectadas en paralelo, este conjunto dará igualmente un voltaje total de 1,5V). La ventaja que logramos es que la duración del sistema manteniendo esa tensión es mayor que si usamos una pila única, debido a que la capacidad (los mAh) del conjunto es la suma total de las capacidades de cada una de las pilas individuales.

Es muy importante asegurarse de que las pilas/baterías conectadas en serie o en paralelo sean del mismo tipo (alcalinas, NiMH, etc.), sean de la misma forma (AA, PP3, etc.), tengan la misma capacidad y aporten el mismo voltaje. Si no se hace así, el funcionamiento del conjunto puede ser inestable e incluso peligroso: en el caso de las baterías LiPo, puede llegar a haber explosiones si no se sigue esta norma. De hecho, en este tipo de baterías se recomienda adquirir packs (en serie o en paralelo) preensamblados, ya que nos ofrecen la garantía de que sus unidades han sido seleccionadas para tener la misma capacidad, resistencia interna, etc., y no causar problemas.

Compra de pilas/baterías

Cualquier tipo de pila/batería que necesitemos en nuestros proyectos (de diferente voltaje, capacidad, composición química...) lo podremos adquirir a través de cualquiera de los distribuidores listados en el apéndice A del libro: solo hay que usar el buscador que ofrecen en sus tiendas online (normalmente introduciendo en él el código numérico del producto deseado) para encontrar la pila/batería deseada.

EL MUNDO GENUINO-ARDUINO

Por ejemplo, el distribuidor online de componentes electrónicos Sparkfun ofrece varias baterías LiPo con los siguientes códigos de producto: el nº **341** (3,7V/850mAh), el nº **339** (3,7V/1Ah), el nº **8483** (3,7V/2Ah) o el nº **8484** (3,7V/6Ah), todas con conector JST de 2 pines. Otro popular distribuidor online de componentes electrónicos es Adafruit, el cual, por su parte, también ofrece varias baterías LiPo, como son los productos nº **1578** (3,7V/500mAh), nº **258** (3,7V/1,2Ah), nº **2011** (3,7V/2Ah), nº **328** (3,7V/2,5Ah)... y también baterías Li-ion, como el producto nº **1781** (3,7V/2,2Ah), el nº **354** (3,7V/4,4Ah) o el nº **353** (3,7V/6,6Ah), todas ellas con conector JST de 2 pines también. Destaquemos, además, que existen otros formatos interesantes, como por ejemplo el producto nº **10053** (9V/350mAh) de Sparkfun, consistente en una batería Li-ion en formato PP3 –notar la elevada tensión que proporciona–, el producto nº **12895** (3,7V/2,6Ah) también de Sparkfun, consistente en otra batería Li-ion pero esta vez en formato 18650 (ambas, eso sí, sin circuito protector –ver apartado "Compra de cargadores"–) o el producto con código **FIT0137** del distribuidor online DFRobot (7,4V/2,2Ah), consistente en una batería LiPo que viene con clavija de 5,5/2,1mm compatible con el zócalo de las placas Arduino.

Si buscamos pilas alcalinas o recargables de tipo NiMH, es incluso más sencillo porque los modelos más habituales están disponibles en cualquier comercio local, aunque si quisieramos adquirirlas a través de Sparkfun o Adafruit, también podemos: por ejemplo, una pila alcalina PP3 aparece como producto nº **10218** en Sparkfun (y nº **1321** en Adafruit), una pila alcalina AAA como producto nº **9274** en Sparkfun (y nº **617** en Adafruit), una pila alcalina AA como producto nº **9100** en Sparkfun, una pila NiMH AA como producto nº **335** en Sparkfun, etc.

También podemos conseguir fácilmente pilas/baterías de tipo "botón". Por ejemplo, Sparkfun distribuye la pila no recargable CR2032 –que tiene una capacidad de 220mAh– como producto nº **338** (Adafruit como producto nº **654**) y la batería recargable LIR2450 –de 120mAh– como producto nº **10319** (Adafruit como producto nº **1572**). Otros modelos disponibles en Adafruit son el CR1220 (pila no recargable –de 40mAh– con código de producto nº **380**), el CR2016 (pila no recargable –de 90mAh– con código de producto nº **2849**) o el CR2450 (pila no recargable –de 610mAh– con código de producto nº **2850**). No todas tienen las mismas dimensiones.

Compra de portapilas (con distintos conectores)

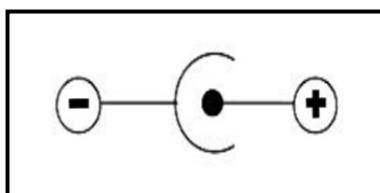
Si se van a emplear pilas alcalinas o recargables NiMH, lo más habitual es utilizar en nuestros proyectos algún tipo de portapilas que permita utilizar varias unidades conectadas en serie. Existen muchos modelos; como muestra, tan solo en la página web del distribuidor de componentes electrónicos Adafruit podemos

encontrar (escribiendo, tal como ya sabemos, su número de producto en el buscador integrado del portal), los siguientes productos:

- Nº 727:** portapilas de 3 unidades AAA con conector JST-PH e interruptor.
- Nº 830:** " de 4 unidades AA con cables-borne al aire e interruptor.
- Nº 248:** " de 6 unidades AA con clavija de 5,5/2,1mm y sin interruptor.
- Nº 449:** " de 8 unidades AA con cables-borne al aire y sin interruptor.
- Nº 875:** " de 8 unidades AA con clavija de 5,5/2,1mm e interruptor.
- Nº 67:** " de 1 unidad PP3 con clavija de 5,5/2,1mm e interruptor.
- Nº 783:** " de 2 unidades CR2032 con conector JST-PH e interruptor.
- Nº 80:** clip para conectar 1 unidad PP3 con clavija de 5,5/2,1mm.

Por "clavija de 5,5/2,1mm" entendemos un tipo de conector formado por un cilindro metálico macizo (de 2,1mm de diámetro) rodeado de un cilindro hueco también metálico (de 5,5mm de diámetro). Este tipo de clavija (también llamada en la literatura anglosajona "barrel plug") es, con diferencia, el más habitual en los adaptadores AC/DC y el zócalo hembra correspondiente (también llamado en la literatura anglosajona "barrel jack") está presente en multitud de dispositivos electrónicos (como por ejemplo las placas Arduino); esto hace que la combinación de clavija y zócalo 5,5/2,1mm sea una de las maneras más comunes de alimentar proyectos electrónicos.

No obstante, mencionando solo "clavija de 5,5/2,1mm" no definimos exactamente el tipo de conector porque queda un detalle pendiente de concretar: ¿cuál de los dos cilindros (el macizo interior o el hueco exterior) es el borne positivo y cuál es el borne negativo?; en el mercado existen ambas posibilidades, pero una abrumadora mayoría son los de tipo "centro positivo"; es decir: el cilindro macizo interior es el borne positivo y el cilindro hueco exterior el negativo. En este libro daremos por supuesto que, al referirnos a un conector 5,5/2,1mm, estaremos hablando siempre de uno de "centro positivo" (ya que es muy poco probable que no sea así) pero si en cualquier momento el lector quisiera saber si la clavija que acompaña cierto producto es "centro positivo", solo tiene que mirar el siguiente diagrama pintado en algún lugar de ese producto:



EL MUNDO GENUINO-ARDUINO

En el caso de utilizar un portapilas con cables al aire, las posibilidades de conexión son muy variadas: una placa Arduino puede ser alimentada directamente así, al igual como cualquier placa de prototipado (tal como estudiaremos en próximos apartados). Incluso, si fuera necesario, podemos utilizar el producto nº 369 de Adafruit para convertir terminaciones al aire en una clavija 5,5/2,1mm. En cambio, si usamos un portapilas terminado en un conector JST-PH, deberemos adquirir y emplear algún componente electrónico extra que contenga un receptáculo de este tipo para hacer de intermediario entre el portapilas y el resto del circuito a alimentar (como por ejemplo los productos de Adafruit nº 1862 y nº 1863 –idéntico al anterior pero con interruptor añadido– o el producto nº 13685 de Sparkfun), ya que ni las placas Arduino ni las placas de prototipado que estudiaremos disponen de ningún zócalo JST-PH.

De forma similar, Sparkfun nos ofrece los siguientes productos:

- Nº 9543: portapilas de 2 unidades AAA con cables-borne al aire e interruptor.
- Nº 9925: " de 2 unidades AA con conector JST-PH e interruptor.
- Nº 9547: " de 2 unidades AA con cables-borne al aire e interruptor.
- Nº 10891: " de 3 unidades AA con cables-borne al aire e interruptor.
- Nº 552: " de 4 unidades AA con cables-borne al aire y sin interruptor.
- Nº 12083: " de 4 unidades AA con cables-borne al aire e interruptor.
- Nº 9835: " de 4 unidades AA con clavija de 5,5/2,1mm y sin interruptor.
- Nº 10512: " de 1 unidad PP3 con clavija de 5,5/2,1mm y sin interruptor.
- Nº 12618: " de 2 unidades CR2032 con cables-borne al aire e interruptor.
- Nº 91: clip para conectar 1 unidad PP3 con conector JST-PH.

DFRobot también ofrece productos similares; destacaremos (por no aparecer en las listas anteriores) el producto con código **FIT0111** (clip para conectar 1 unidad PP3 con cables-borne al aire), los productos **FIT0078** y **FIT0266** (portapilas de 6 unidades AA con cables-borne al aire sin interruptor) o el producto **FIT0362** (portapilas de 3 unidades AA con conector USB micro-B sin interruptor), entre otros.

Compra de cargadores

Si en nuestros proyectos empleamos baterías (recargables, se entiende), otro elemento que no debemos obviar es un cargador. Eso sí, dependiendo del tipo de batería (NiMH, LiPo, etc.), deberemos elegir un determinado tipo de cargador u otro.

Por ejemplo, para las baterías de tipo NiMH (ya sean en formato AA, AAA o PP3) nos será suficiente con el ofrecido, por ejemplo, por el distribuidor generalista

CAPÍTULO 1: ELECTRÓNICA BÁSICA

MinilntheBox con código de producto **nº 194037** (o también **nº 440658**, entre otros); estos productos se conectan directamente a un enchufe de pared y no requieren mayor atención.

En cambio, para recargar baterías LiPo (o Li-Ion), no nos vale cualquier cargador porque hay que tener la precaución de usar siempre uno que cumpla dos condiciones: que someta la batería a un voltaje (el "voltaje de carga") menor o igual (lo preferible) que el voltaje nominal máximo de esa batería, y que, además, aporte a la batería una intensidad (la "intensidad de carga") menor (lo preferible, aunque el proceso de carga tarde más en completarse) o igual que la capacidad (1C) de dicha batería. Si no se cumplen las dos condiciones anteriores, el cargador podría dañar la batería irreversiblemente (e incluso hacerla explotar).

De hecho, las baterías LiPo son muy delicadas: también se corre el riesgo de explosión cuando se descargan por debajo de su voltaje de corte (normalmente 3V), o cuando son obligadas a aportar más corriente de la que pueden ofrecer (normalmente 2C), o cuando son utilizadas en ambientes de temperaturas extremas (normalmente fuera del rango 0º-50ºC), entre otras causas. Por eso muchas de estas baterías (aunque no todas) incorporan un circuito protector que detecta estas situaciones y desconecta la batería de forma segura. De todas formas, para conocer las características específicas de cada batería (como las tensiones, intensidades y temperaturas seguras) es obligatorio consultar la información que ofrece el fabricante para esa batería en concreto (su datasheet).

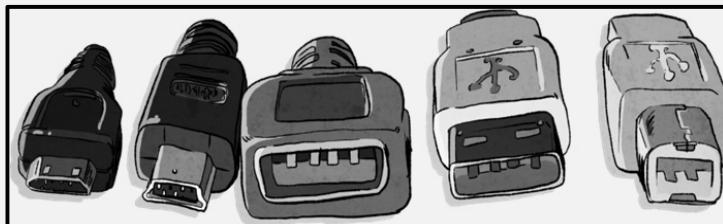
Podemos encontrar cargadores LiPo (y Li-ion) de muchos tipos y características. Por ejemplo, en la página web de Sparkfun podemos encontrar, entre otros, los productos:

Nº 10217: cargador para baterías LiPo/Li-Ion que aporta un voltaje de carga de 3,7V y una intensidad de carga de 500mA. Consiste en una plaquita que dispone por un lado de un zócalo JST-PH (donde se podrá conectar la batería a cargar) y por otro lado de un zócalo USB de tipo micro-B (donde se deberá conectar una fuente de alimentación externa que ha de proporcionar obligatoriamente una tensión de 5V y una intensidad de, al menos, igual a la de carga; en el siguiente apartado mencionaremos varias fuentes que cumplen estos requisitos).

NOTA: Gracias al chip controlador MCP73831 que esta plaquita lleva incorporado (chip que, por otro lado, viene integrado en la mayoría de cargadores que mencionaremos aquí), el voltaje aportado por la fuente externa (5V) es rebajado convenientemente al voltaje de carga óptimo (3,7V) para las baterías LiPo/Li-ion.

Breve nota sobre los conectores y el protocolo USB

Cuando estamos hablando de "conector USB", tenemos que tener claro de qué tipo de conector estamos hablando, porque existen varios modelos. En la imagen siguiente se muestran algunos de los más extendidos; de izquierda a derecha: conector micro-B macho, conector mini-B macho (actualmente obsoleto), conector A hembra, conector A macho y conector B macho.



En una conexión USB entre varios dispositivos siempre ha de existir obligatoriamente uno actuando como "maestro" (el llamado "host") y otro –u otros– actuando como "esclavos" (los llamados "periféricos"). El dispositivo "host" es el único que puede iniciar y controlar la transferencia de datos entre el resto de dispositivos conectados, mientras que los "periféricos" tan solo pueden responder a las peticiones hechas por el "host" y poca cosa más. Además, el dispositivo "host" es el encargado de tener que alimentar eléctricamente a todos los "periféricos" conectados a él (ofreciendo una tensión de 5V y aportando como máximo 500mA). Esto significa que el protocolo USB en realidad no solo es un protocolo de transferencia de datos, sino también un estándar de alimentación eléctrica.

Los dispositivos "host" normalmente disponen de un zócalo USB de tipo A (conector "hembra") donde los dispositivos periféricos se enchufarán mediante el correspondiente conector USB macho de tipo A. Algunos dispositivos periféricos, por su parte, también puede que dispongan de un zócalo USB, pero en este caso este deberá ser de tipo B, mini-B o micro-B. Un "host" típico es un computador, al cual se le pueden conectar varios periféricos, como teclados, ratones, lápices de memoria, cámaras de fotos o vídeo, teléfonos móviles de última generación, etc.

Nº 12711: cargador para baterías LiPo/Li-Ion en forma de plaquita que también aporta un voltaje de carga de 3,7V y una intensidad de carga de 500mA, pero que, a diferencia del producto anterior, además de ofrecer un zócalo USB micro-B también ofrece un zócalo de 5,5/2,1mm para poder elegir dónde conectar la fuente de alimentación externa (la cual ha de proporcionar obligatoriamente una tensión estable de 5V y una intensidad, al menos, igual a la de carga; en el siguiente apartado mencionaremos varias fuentes AC/DC

que cumplen estos requisitos) y, por el otro lado, ofrece dos zócalos JST-PH: uno etiquetado como "BATT-IN" (donde se podrá conectar la batería a cargar) y otro etiquetado como "SYS OUT" (donde se podrá conectar un circuito cualquiera, de manera que la misma batería que esté conectada en "BATT-IN" pueda ser utilizada directamente como fuente de alimentación para ese circuito.

DFRobot distribuye un cargador (con código de producto **DFR0208**) muy parecido al nº 10217 de Sparkfun. Adafruit, por su lado, distribuye otro cargador (con código de producto **nº 1904**) también muy similar pero con una diferencia: su intensidad de carga es de 100mA (en vez de los 500mA de los anteriores), haciéndolo, por tanto, compatible con más baterías. Freetronics distribuye asimismo un cargador muy parecido a este último (aunque el zócalo JST-PH no viene soldado por defecto) con el nombre de "**USB LiPo Charger**".

Otro cargador-plaquita interesante distribuido por Adafruit es el **nº 259**, el cual también aporta 3,7V y –en este caso– una intensidad máxima de carga de 500mA; lo interesante de este cargador está en que no solo dispone, como los productos mencionados en el párrafo anterior, del zócalo USB micro-B para enchufar allí la fuente de alimentación externa (que ha de estar regulada a 5V y que ha de ser capaz, recordemos, de proporcionar al menos la intensidad máxima de carga) y del zócalo JST-PH (etiquetado como "BATT") para conectar la batería a cargar, sino que también dispone –como el producto nº 12711 de Sparkfun anteriormente descrito– de otro zócalo JST-PH (etiquetado como "LOAD") donde se puede conectar el circuito que se desee alimentar mientras tanto; hay que tener en cuenta que de lo que no dispone este cargador es de zócalo 5,5/2,1mm.

Finalmente, otro cargador-plaquita interesante también distribuido por Adafruit es su producto **nº 1304**, el cual aporta 3,7V a 100mA y consta, además del zócalo JST-PH para conectar la batería a cargar, de una clavija USB macho de tipo A (he aquí la novedad) diseñada para recibir la alimentación externa directamente desde un zócalo USB tipo A como los que están presentes en cualquier computador. Siguiendo la misma filosofía de poder enchufar directamente el cargador a un zócalo USB de nuestro computador, no está de más mencionar el producto **nº 1573** de Adafruit, el cual permite recargar baterías LIR2450 (Li-ion de tipo "botón").

Características de los adaptadores AC/DC

El otro tipo de fuente de alimentación externa, diferente de las pilas/baterías, que más utilizaremos para nuestros circuitos es el adaptador AC/DC. Su función típica es conectarse a una toma de la red eléctrica general (un "enchufe") para transformar

EL MUNDO GENUINO-ARDUINO

el elevado voltaje alterno ofrecido por ella (en España es de $230V \pm 5\%$ y $50Hz \pm 0,3\%$; para conocer el de otros países se puede consultar el artículo titulado "Enchufes, voltajes y frecuencias" de la Wikipedia) en un voltaje continuo (y mucho menor) para ofrecer entonces este a los aparatos que se le conecten y así ponerlos en funcionamiento de una forma estable y segura.

Los adaptadores AC/DC básicamente están formados por un circuito transformador, el cual convierte el voltaje AC de entrada en otro voltaje AC mucho menor, y un circuito rectificador, el cual convierte ese voltaje AC ya transformado en un voltaje DC, que será el voltaje final de salida. Todos los adaptadores incorporan una etiqueta impresa que informa tanto del rango de valores en el voltaje AC de entrada con el que son capaces de trabajar (además de la frecuencia de la señal AC admitida) como del valor del voltaje DC y de la intensidad máxima que ofrecen como salida. Por ejemplo, la imagen siguiente corresponde a un adaptador AC/DC que admite voltajes AC de entrada entre 100V y 240V a una frecuencia de 50 o 60Hz (por tanto, compatible con la red eléctrica española) y aporta un voltaje DC de salida de 9V y una intensidad máxima de 1A.



Podemos clasificar los adaptadores según si son "regulados" (es decir, si incorporan un regulador de tensión en su interior) o no. Un regulador de tensión es un componente que, estando sometido a un determinado voltaje de entrada relativamente fluctuante, es capaz de generar un voltaje de salida (normalmente menor) mucho más estable, constante y controlado. Esto hace que los adaptadores regulados proporcionen un voltaje de salida muy concreto: precisamente el mostrado en su etiqueta. Lo que sí variará (hasta el máximo mostrado también en la etiqueta) es la intensidad de corriente ofrecida, ya que esta depende en cada momento de las necesidades del circuito alimentado.

Los adaptadores no regulados, en cambio, no poseen ningún mecanismo de estabilización, por lo que proporcionan un voltaje de salida cuyo valor puede llegar a

ser diferente en varios voltios al mostrado en la etiqueta. Este tipo de adaptadores ciertamente reducen el voltaje de entrada a un valor de salida menor, pero el valor concreto de este voltaje de salida dependerá en buena parte del consumo eléctrico realizado en ese momento particular por el circuito alimentado (o dicho de otra forma, de la intensidad de corriente transferida a dicho circuito en cada instante).

Expliquemos esto: debido al diseño y construcción interna de los adaptadores no regulados, ocurre que a medida que el circuito consume más intensidad de corriente, el voltaje de salida ofrecido por el adaptador (inicialmente bastante más elevado que el valor nominal marcado en su etiqueta) se va reduciendo cada vez más hasta llegar a su valor nominal cuando el circuito consume la máxima intensidad que el adaptador es capaz de ofrecer, (cuyo valor es el indicado en la etiqueta impresa). Si el circuito sigue aumentando su consumo y supera esa intensidad máxima, el voltaje ofrecido por el adaptador seguirá disminuyendo y llegará a ser menor que el nominal, circunstancia en la que se corre el riesgo de dañar el adaptador (y de rebote, el circuito alimentado). Es fácil comprobar este comportamiento con un multímetro, tal como veremos en un apartado posterior de este mismo capítulo.

Por otro lado, otro inconveniente de los adaptadores no regulados (causado por su falta de precisión a la hora de rectificar la señal AC original) es la carencia de estabilidad en el valor del voltaje de salida obtenido, sea cual sea este. Así pues, es común observar variaciones periódicas alrededor de un determinado valor "nominal" (es decir, oscilaciones) cuya magnitud puede llegar a ser de hasta varios voltios. Desgraciadamente, es difícil observar este fenómeno con un multímetro porque este tipo de aparatos generalmente muestra valores medios para un intervalo de tiempo, así que en este caso deberíamos usar una herramienta más sofisticada como es un oscilloscopio. Si el lector quiere profundizar más sobre este tema, puede encontrar una buena introducción en <http://www.teamwavelength.com/info/powersupply.php>.

La principal razón de la existencia de los adaptadores no regulados es su precio: son más baratos. No obstante, en este libro recomendaremos el uso de adaptadores regulados por su mayor fiabilidad y seguridad. Concretamente, para alimentar la mayoría de placas Arduino es ideal el producto **nº63** de Adafruit, un adaptador regulado compatible con la red eléctrica española que genera un voltaje de salida estable de 9V y una corriente de magnitud variable según demanda pero de 1A como máximo (además de tener una clavija de 5,5/2,1mm, tal como es el zócalo de alimentación de las placas Arduino). También nos podría servir su producto **nº 798**, un adaptador regulado (también con clavija de 5,5/2,1mm) capaz de generar 12V y (como máximo) 1A. De hecho, cualquier adaptador AC/DC con clavija 5,5/2,1mm que pueda generar una tensión regulada entre 7V y 12V y aportar una

EL MUNDO GENUINO-ARDUINO

corriente de hasta 1A (más es superfluo porque las placas Arduino no consumen tanto) será compatible con las placas Arduino más habituales. Sabiendo esto, incluso podríamos optar por elegir el producto **nº 1448** de Adafruit, un adaptador regulado cuya clavija puede intercambiarse entre varias a elegir (entre las cuales está la de 5,5/2,1mm) y cuyo voltaje de salida puede ser ajustado entre varios valores posibles (siendo 7,5V, 9V o 12V los más útiles para nosotros, con intensidades máximas de 0,9A, 0,8A y 0,7A, respectivamente). Por otro lado, Sparkfun también distribuye otros adaptadores regulados con clavija de 5,5/2,1mm recomendables, como son los productos **nº298** y **nº 9442**, generadores de 9V y (como máximo) 650mA y 12V y (como máximo) 600mA, respectivamente.

En el caso de querer alimentar nuestra placa Arduino a través de su zócalo USB en vez de su zócalo de 5,5/2,1mm (cosa que también es posible, tal como veremos en el siguiente capítulo), necesitaremos un adaptador AC/DC que, además de ofrecer ese tipo de conexión, genere una tensión regulada que sea específicamente de 5V (esto es debido a ciertas características eléctricas internas de las placas Arduino que ya estudiaremos). En este sentido nos pueden ser útiles el producto **nº 501** de Adafruit o el **nº 11456** de Sparkfun, los cuales ofrecen también 1A como máximo (aunque, por diseño, en realidad las placas Arduino tan solo pueden consumir mediante su zócalo USB hasta un máximo 500mA más allá del cual la alimentación se interrumpirá por seguridad) en combinación con un cable USB (A macho<->B macho, tal como los productos **nº 900** o **nº 62** de Adafruit, o el **nº 512** de Sparkfun).

Estos cables USB, de hecho, también podrían ser conectados a un zócalo USB tipo A de nuestro propio computador y así conseguir alimentar la placa Arduino a partir de él (la conexión USB placa-computador también está dentro de los márgenes eléctricos compatibles con las placas Arduino). O también podríamos conectarlos al zócalo USB de tipo A que ofrecen diversas fuentes de alimentación portátiles recargables (como los productos **nº 1959** o **nº 1565** de Adafruit –que también funcionan a 5V y 1A máx.– o el producto **nº 11358** o **nº 11360** de Sparkfun, entre otros). Un cable algo más exótico es el producto **nº 8639** de Sparkfun (o el **nº 2697** de Adafruit, es lo mismo), ya que por un lado ofrece un enchufe USB tipo A y por otro una clavija de 5,5/2,1mm, permitiendo así alimentar las placas Arduino mediante todas las fuentes y los adaptadores anteriores pero a través de su zócalo de 5,5/2,1 mm.

Por otro lado, los cargadores de baterías LiPo descritos en el apartado anterior también pueden ser alimentados con los adaptadores AC/DC **nº 501** (Adafruit) o **nº 11456** (Sparkfun) recién mencionados, pero el cable USB empleado deberá ser en este caso de tipo A macho<->microB macho, como el producto **nº 592**.

CAPÍTULO 1: ELECTRÓNICA BÁSICA

(Adafruit) o el nº **10215** (Sparkfun). De igual manera, se podría utilizar una conexión USB de nuestro computador o, también, cualquier alimentador de teléfono móvil de última generación.

En el caso particular de querer cargar la batería LiPo FIT0137 de DFRobot (recordemos que proporcionaba 7,4V/2,2Ah; esto es, el doble de voltaje que el resto de baterías mencionadas) podemos echar mano de un cargador AC/DC distribuido por la propia DFRobot con código FIT0398, el cual dispone –de forma intercambiable– tanto de un zócalo de 5,5/2,1mm para conectarle dicha batería como de una clavija del mismo calibre para alimentar, si así lo quisieramos, nuestra placa Arduino (ya que proporciona un voltaje regulado de 8,5V –y 1,5A como máximo–).

Otros productos complementarios que nos podrán ser útiles en determinados proyectos son, por ejemplo, el nº **1125** de Adafruit (un interruptor diseñado para enchufarse en serie a la clavija 5,5/2,1mm de un adaptador AC/DC y así disponer de un mecanismo sencillo para abrir el circuito de alimentación), el nº **1351** de Adafruit (un cable divisor de alimentación eléctrica con un zócalo 5,5/2,1mm de entrada y dos clavijas del mismo calibre de salida), el nº **327** de Adafruit o el **FIT0114** de DFRobot (un alargador de la clavija 5,5/2,1mm), el nº **990** (un enchufe–adaptador de pared para el estándar mayoritario en Europa y América del Sur, necesario ya que el enchufe de los productos de Adafruit sigue el estándar anglosajón), etc.

También hay que tener presente que si nuestros circuitos necesitaran una alimentación que aportara más voltaje o intensidad del que son capaces de suministrar los adaptadores AC/DC mencionados en los párrafos anteriores (porque en nuestro circuito tenemos dispositivos –como por ejemplo muchos tipos de motores– que tienen un mayor consumo), existen varias alternativas: tenemos, por ejemplo, el producto nº **352** de Adafruit (que ofrece un voltaje de salida de 12V y corriente máxima de 5A) o el nº **658** (5V y 10A máx.), entre muchos otros. Hay que tener siempre la precaución, no obstante, de no utilizar un adaptador que ofrezca un voltaje de salida mayor que el circuito sea capaz de admitir o una intensidad máxima menor que el circuito pueda llegar a demandar para no dañar ni el circuito ni el propio adaptador.

Breve nota sobre las fuentes de alimentación solares

Otra manera de alimentar eléctricamente nuestros circuitos y/o también de recargar nuestras baterías (generalmente de tipo LiPo/Li-ion) es mediante el uso de paneles fotovoltaicos, una posibilidad muy interesante para proyectos que necesiten ser implementados en el exterior (fuera del alcance de la red eléctrica doméstica) y que requieran cierto grado de autonomía. En este sentido, la empresa

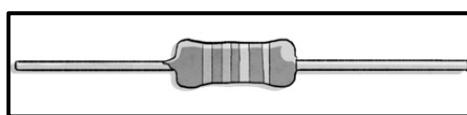
Voltaic Systems (<http://www.voltaicsystems.com>) ofrece una amplia variedad de productos llamados "Solar Charger Kits" (diferentes entre sí sobre todo en la potencia aportada) que permiten alimentar a una placa Arduino (y, a través de esta, a todo el circuito que se le conecte) por su conector USB, simplemente. En realidad, estos productos básicamente constan de una batería recargable mediante un panel solar asociado, el cual es capaz de alimentar a la placa Arduino mientras va siendo recargada pero también (si ya ha conseguido suficiente carga) en los momentos donde no hay suficiente luz.

Una opción alternativa es la plaquita **Lipo Rider Pro** de Seeedstudio, la cual permite cargar baterías LiPo de 3,7V (a 500mA de intensidad de carga) conectadas a ella vía JST-PH a partir de un panel solar (el propio Seeestudio ofrece unos cuantos a elegir) conectado también a ella a través de otro zócalo JST a la vez que puede estar alimentando en paralelo una placa Arduino conectada a ella a través de su zócalo USB de tipo A. Un cargador similar (que, eso sí, en vez de zócalo USB ofrece cables-borne) es el **DFR0264** de DFRobot.

También podemos usar el producto **nº 390** de Adafruit, un cargador de baterías LiPo de 3,7V que permite obtener la carga (con una intensidad máxima de 500mA) desde una fuente externa regulada a 5V (bien a través de un zócalo 5,5/2,1mm o bien a través de un zócalo USB mini-B –en este sentido, funcionaría igual que el producto **nº 259** mencionado párrafos atrás–) pero también, alternativamente, desde un panel solar que aporte como máximo 6V (tales como los productos **nº 200, nº 417, nº 500 y nº 1525** del propio Adafruit, o los **nº 9241 y nº 7840** de Sparkfun). Al igual que el ya mencionado producto **nº 259**, este cargador dispone de un zócalo JST etiquetado como "BATT" para conectar la batería a cargar y otro etiquetado como "LOAD" donde se puede conectar el circuito que se desee alimentar mientras tanto.

COMPONENTES ELÉCTRICOS

Resistencias



Un resistor o resistencia es un componente electrónico utilizado simplemente para añadir, como su nombre indica, una resistencia eléctrica entre dos puntos de un circuito. De esta manera, y gracias a la Ley de Ohm, podremos distribuir según nos convenga diferentes tensiones y corrientes a lo largo de nuestro circuito.

Debido al pequeño tamaño de la mayoría de resistores, normalmente no es posible serigrafiar su valor sobre su encapsulado, por lo que para conocerlo debemos saber interpretar una serie de líneas de colores dispuestas a lo largo de su cuerpo. Normalmente, el número de líneas de colores son cuatro, siendo la última de color dorado o bien plateado (aunque puede ser de otros colores también, siendo en ese caso la línea más separada de las demás). Esta línea dorada o plateada indica la tolerancia de la resistencia, es decir: la precisión de fábrica que esta nos aporta. Si es de color dorado indica una tolerancia del $\pm 5\%$ y si es plateada una del $\pm 10\%$ (otros colores indican otros valores menos habituales: el rojo un $\pm 2\%$, el marrón un $\pm 1\%$, etc.); en cualquier caso, cuanto menor sea ese valor, mayor será el precio de la resistencia). Por ejemplo, una resistencia de 220Ω con una franja plateada de tolerancia, tendría un valor posible entre 198Ω y 242Ω (es decir, $220\Omega \pm 10\%$). Este dato se ha de tener en cuenta especialmente en el diseño de circuitos con resistencias conectadas en serie o en paralelo para controlar que los valores mínimo y máximo de la resistencia equivalente total estén dentro de los márgenes aceptados en nuestro proyecto.

Las otras tres líneas de colores indican el valor nominal de la resistencia. Para interpretar estas líneas correctamente, debemos colocar a nuestra derecha la línea de tolerancia, y empezar a leer de izquierda a derecha, sabiendo que cada color equivale a un dígito diferente (del 0 al 9). La primera y segunda línea las tomaremos cada una como el dígito tal cual (uno seguido del otro) y la tercera línea representará la cantidad de ceros que se han de añadir a la derecha de los dos dígitos anteriores. La tabla para conocer el significado numérico de los posibles colores de una resistencia es la siguiente:

Color de la banda	Valor equivalente (en la 1 ^a y 2 ^a banda)	Multiplicador (valor de la 3 ^a banda)
Negro	0	$\times 10^0 = 1$
Marrón	1	$\times 10^1 = 10$
Rojo	2	$\times 10^2 = 100$
Naranja	3	$\times 10^3 = 1000$
Amarillo	4	$\times 10^4 = 10000$
Verde	5	$\times 10^5 = 100000$
Azul	6	$\times 10^6 = 1000000$
Violeta	7	$\times 10^7 = 10000000$
Gris	8	$\times 10^8 = 100000000$
Blanco	9	$\times 10^9 = 1000000000$

EL MUNDO GENUINO-ARDUINO

Por ejemplo, si tenemos una resistencia con las líneas de colores "rojo-verde-naranja", podremos consultar la tabla para deducir que tendremos una resistencia de $25 \cdot 1000 = 25 \text{ K}\Omega$. Otro ejemplo: si tenemos una resistencia con las líneas de colores "marrón-negro-azul", tendremos entonces una resistencia de $10 \cdot 1000000 = 10 \text{ M}\Omega$.

También nos podemos encontrar con resistencias que tengan cinco líneas impresas: en ese caso, su interpretación es exactamente igual, solo que en vez de dos disponemos de tres líneas para indicar los tres primeros dígitos del valor de la resistencia, siendo la cuarta la que representa el multiplicador y la quinta la tolerancia. Algunas resistencias incluso tienen hasta seis líneas impresas (son las más precisas, pero en nuestros proyectos pocas veces las necesitaremos); en ese caso, lo único que cambia es que aparece una sexta línea a la derecha de la línea de la tolerancia indicando un nuevo dato: el coeficiente de temperatura de la resistencia, el cual nos informa sobre cuánto varía el valor de esa resistencia dependiendo de la temperatura ambiente (medida en ppm/°C, donde 10000ppm=1%).

Ha de quedar claro, en todo caso, que aunque para conocer el orden de las franjas y leer el valor de una resistencia hemos de colocar esta en un sentido determinado, los resistores no tienen polaridad. Esto quiere decir que al incluirlo en un circuito es indiferente conectar sus dos terminales en un sentido o del revés.

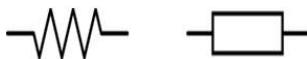
Por otro lado, además de conocer el valor resistivo que aportan estos componentes, también hemos de tener en cuenta la intensidad de corriente que pueden soportar como máximo sin fundirse. Para ello, el fabricante nos deberá proporcionar siempre un dato: la potencia máxima que la resistencia es capaz de disipar en forma de calor, valor que está directamente relacionado con su tamaño. Las resistencias más utilizadas en la electrónica son las de 1/4W, 1/2W y 1W (siendo la de 1/4W la más pequeña y la de 1W la más grande de las tres). En este sentido, para conocer qué tipo de resistencia nos interesará utilizar en nuestros circuitos, debemos utilizar la fórmula ya conocida de $P=V \cdot I$ (donde V es la diferencia de potencial presente entre los extremos de la resistencia e I es la corriente que circula por ella), o bien alguna de las fórmulas equivalentes, como $P=I^2 \cdot R$ o $P=V^2/R$ (donde R es el valor de la resistencia propiamente dicha). De cualquiera de estas maneras, obtendremos la potencia que debe ser capaz de disipar nuestra resistencia, con lo que ya tendremos el criterio para elegirla. No es mala idea utilizar una resistencia cuyo poder disipador sea aproximadamente el doble del resultado obtenido para no sufrir posibles sobrecalentamientos.

Para no ir escasos de resistencias en nuestros proyectos, lo más recomendable es adquirir (en nuestro proveedor de componentes electrónicos

favorito) paquetes ya preparados de resistores que incluyen cantidades variadas de ellos con diversos valores resistivos. Ejemplos de ello son el producto nº **10969** de Sparkfun o el producto con código **FIT0119** de DFRobot, por citar un par.

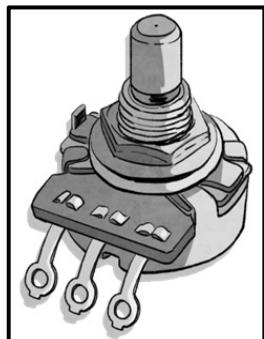
Finalmente, es conveniente saber, aunque no las necesitaremos en los proyectos mostrados en este libro, que existe otro tipo de resistencias (de tamaño muy reducido), que está especialmente diseñado para ser soldado directamente a la superficie de una placa de circuito impreso. Estos resistores utilizan, en lugar de colores, una secuencia de dígitos, que serán tres o cuatro dependiendo de si su tolerancia es del 5% o del 1%, respectivamente. En el primer caso, las dos primeras cifras indican el valor de la resistencia y la tercera, su multiplicador (por ejemplo, el valor "205" indicará una resistencia de $2M\Omega - 20 \times 10^5$). En el segundo caso, las tres primeras cifras indican el valor de la resistencia y la cuarta, su multiplicador. También podremos encontrarnos con resistencias que tienen imprimida sobre su superficie dos cifras seguidas de una letra (como por ejemplo, "01C"): en este caso, para conocer su valor resistivo concreto no tendremos más remedio que consultar (por ejemplo, aquí: <http://www.hobby-hour.com/electronics/smdcalc.php>) la tabla "EIA-96", internacionalmente estandarizada.

Los símbolos utilizados en el diseño de los circuitos eléctricos para representar una resistencia pueden ser dos:



donde el de la derecha es el estándar normalizado por la "International Electrotechnical Commission" (IEC), aunque el de la izquierda sigue siendo ampliamente utilizado actualmente.

Potenciómetros

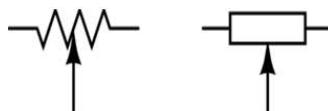


Un potenciómetro es una resistencia de valor variable. Podemos darnos cuenta de su gran utilidad con un ejemplo muy simple: si suponemos que tenemos una fuente de alimentación que genera un determinado voltaje estable y, tenemos presente la Ley de Ohm ($V=I \cdot R$), podemos ver que si aumentamos de valor la resistencia R , a igual voltaje la intensidad de corriente que pasará por el circuito inevitablemente disminuirá. Y al contrario: si disminuimos el valor de R , la corriente I aumentará. Si esta variación de R la podemos controlar nosotros a voluntad, podremos alterar como queramos la corriente que circula por un circuito. De hecho, un uso muy

EL MUNDO GENUINO-ARDUINO

habitual de los potenciómetros es el de hacer de divisores de tensión progresivos, con lo que podremos, por poner un ejemplo, encender o apagar paulatinamente una luz a medida que vayamos cambiando el valor de R.

Un potenciómetro suele disponer físicamente de tres patillas: entre las dos de sus extremos existe siempre un valor fijo de resistencia (el máximo, de hecho), y entre cualquiera de esos extremos y la patilla central tenemos una parte de ese valor máximo. Es decir: la resistencia máxima que ofrece el potenciómetro entre sus dos extremos no es más que la suma de las resistencias entre un extremo y la patilla central (llamémosla R1), y entre la patilla central y el otro extremo (llamémosla R2). De aquí se puede pensar que un potenciómetro es equivalente a dos resistencias en serie, pero la gracia está en que en cualquier momento podremos modificar el estado de la patilla central para conseguir aumentar la resistencia de R1 (disminuyendo como consecuencia la resistencia R2, ya que el valor total máximo sí que permanece constante) o bien al contrario, para conseguir disminuir la resistencia de R1 (aumentando por lo tanto la resistencia R2 automáticamente). De hecho, los símbolos que se utilizan en el diseño de circuitos electrónicos para representar un potenciómetro recuerdan esta conformación:



La manera concreta de alterar el estado de la patilla central del potenciómetro puede variar y suele depender de su encapsulamiento físico, pero por lo general suele consistir en el desplazamiento de un cursor manipulable conectado a dicha patilla. Podemos encontrarnos potenciómetros de movimiento rotatorio como los del control de volumen de la mayoría de altavoces, o de movimiento rectilíneo como los que se utilizan en las mesas de mezcla de sonido, entre otros. En la imagen mostrada anteriormente se puede ver uno de movimiento rotatorio.

La clasificación más interesante, no obstante, viene a la hora de distinguir el comportamiento que tiene un potenciómetro en el momento que modificamos el estado de su patilla central (es decir, cuando "es movida"). Si el potenciómetro tiene un comportamiento llamado "lineal", la alteración del valor de su resistencia es siempre directamente proporcional al recorrido de la patilla central: es decir, si desplazamos por ejemplo la patilla un 30%, se aumentará/disminuirá la resistencia un 30% también. Por el contrario, si el potenciómetro tiene un comportamiento "logarítmico", la alteración del valor de su resistencia será muy leve al principio del recorrido de la patilla (y por tanto, habrá que realizar un gran desplazamiento de esta para obtener un cambio apreciable de resistencia) pero a medida que se siga

realizando más recorrido de la patilla, la alteración de la resistencia cada vez será proporcionalmente mayor y mayor, hasta llegar a un punto donde un leve desplazamiento producirá una gran cambio en la resistencia. Los potenciómetros logarítmicos son empleados normalmente para el audio, ya que el ser humano no oye de manera lineal: para experimentar por ejemplo una sensación acústica de "el doble de fuerte", es necesario que el volumen físico del sonido sea unas diez veces mayor.

En los proyectos mostrados a lo largo de este libro se usarán potenciómetros lineales, siendo el resto de sus especificaciones no demasiado importantes. Ejemplos válidos de este tipo de potenciómetros pueden ser el producto **nº 9288** de Sparkfun o el **nº 562** de Adafruit (ambos con 10KΩ de resistencia máxima) o el **nº 1789** y **nº 1831** de Adafruit (el primero con 1KΩ y el segundo con 100KΩ de resistencia máxima). Si prevemos que, tras su calibración inicial, no va a ser necesario ajustar más veces algún potenciómetro de nuestro circuito, podemos utilizar entonces los llamados "trimpots", como el producto **nº 9806** de Sparkfun o el **nº 356** de Adafruit (ambos con 10KΩ de resistencia máxima) o el kit **FIT0189** de DFRobot, el cual incluye varios con diferentes valores de resistencia máxima). Todos ellos son de movimiento rotatorio.

Breve nota sobre los "softpots" o potenciómetros de "membrana"

Un "softpot" o potenciómetro de "membrana" es un tipo de potenciómetro en forma de tira muy delgada. Si a lo largo de esa tira se ejerce presión, la resistencia cambia prácticamente de forma lineal desde unos 100 ohmios hasta unos 10KΩ. Esto permite calcular con gran precisión la posición relativa en la tira de, por ejemplo, nuestro dedo. Por ello, estos dispositivos se suelen utilizar mucho en aparatos domésticos, como reproductores "mp3", televisores, etc.

Un "softpot" dispone de tres pines: el pin de más a la derecha (mirándolo de frente con los pines en la zona inferior) suele ser el de la alimentación, el de más a la izquierda es el de tierra y el central se corresponde a la señal obtenida como lectura analógica. Si se presiona en la zona superior de la tira, obtendremos por ese pin central una lectura de 0, y si vamos presionando hacia abajo llegaremos a obtener una lectura máxima de 1023.

Ejemplos de "softpot" son el producto el **nº 178** de Adafruit (con una longitud de 10cm) o los productos **nº 8679**, **nº 8680** y **nº 8681** de Sparkfun (con una longitud de 20cm, 5cm y 50cm, respectivamente). También podemos encontrar "softpots" circulares, como el producto el **nº 1069** de Adafruit.

EL MUNDO GENUINO-ARDUINO

También existen potenciómetros de tipo digital: estos son chips que constan de diferentes patillas a través de las que se puede controlar mediante pulsos eléctricos el valor concreto de resistencia deseado (entre un determinado mínimo y máximo posible). Un ejemplo es el componente DS1669 de Maxim.

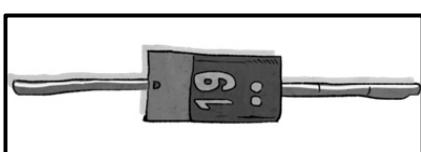
Otras resistencias de valor variable

Los potenciómetros son resistencias que cambian su valor según nuestra voluntad. Pero también existen resistencias que cambian su valor según condicionantes ambientales externos.

Por ejemplo, los fotorresistores (también llamados LDRs –del inglés "Light Dependent Resistor"– o también "celdas CdS" –por el material con el que habitualmente están fabricadas: sulfuro de cadmio–) son resistencias que varían según la cantidad de luz que incide sobre ellos, por lo que se pueden utilizar como sensores de luz. Otro ejemplo son los termistores: resistencias que cambian su valor según varíe la temperatura ambiente, por lo que se pueden utilizar como sensores de temperatura. Otro ejemplo son los sensores de fuerza/presión (también llamados FSRs –del inglés "Force-Sensing Resistor"–), que son resistencias cuyo valor depende de la fuerza/presión a la que son sometidas, o los sensores de flexión: resistencias cuyo valor varía según lo que sean dobladas físicamente, etc.

Si lo que se desea es conocer el símbolo esquemático correspondiente a un tipo concreto de resistencia variable (LDR, termistor, etc.), recomiendo consultar la siguiente página web, la cual incluye la mayoría de símbolos electrónicos existentes, clasificados y ordenados por categorías: <http://www.simbologia-electronica.com>.

Diodos (y LEDs)



El diodo es un componente electrónico con dos extremos de conexión (o "terminales") que permite el paso libre de la corriente eléctrica solamente en un sentido, bloqueándolo si la corriente fluye en el sentido contrario. Este hecho hace que el diodo tenga dos posiciones posibles: a favor de la corriente (llamada "polarización directa") o en contra ("polarización inversa"). Por tanto, a la hora de utilizarlo en nuestros circuitos, debemos de tener en cuenta que la conexión de sus dos terminales se realice en el sentido deseado. Normalmente, los fabricantes nos indicarán cuál es el terminal que ha de conectarse al polo negativo (suponiendo polarización directa) mediante una marca visible cerca de este pintada en el cuerpo del diodo. En la imagen mostrada, esta marca es la gruesa franja blanca a la derecha

hecho hace que el diodo tenga dos posiciones posibles: a favor de la corriente (llamada "polarización directa") o en contra ("polarización inversa"). Por tanto, a la hora de utilizarlo en nuestros circuitos, debemos de tener en cuenta que la conexión de sus dos terminales se realice en el sentido deseado. Normalmente, los fabricantes nos indicarán cuál es el terminal que ha de conectarse al polo negativo (suponiendo polarización directa) mediante una marca visible cerca de este pintada en el cuerpo del diodo. En la imagen mostrada, esta marca es la gruesa franja blanca a la derecha

CAPÍTULO 1: ELECTRÓNICA BÁSICA

del cuerpo del diodo, por lo que el "terminal negativo" será, en este caso, el de la derecha. Técnicamente (siempre suponiendo polarización directa) a ese "terminal negativo" se le llama "cátodo", y al "terminal positivo" se le llama "ánodo".

El diodo se puede utilizar para muchos fines: un uso común es el de rectificador (para convertir una corriente alterna en continua... aunque para ello se necesitan varios diodos y condensadores), pero en nuestros circuitos lo usaremos sobre todo como un elemento suplementario conectado a algún otro componente para evitar que este se dañe si la alimentación eléctrica se conecta por error con la polaridad al revés o bien si esta se encuentra en estado transitorio. A la hora de elegir qué diodos usar en nuestros proyectos, debemos tener en cuenta ciertos aspectos importantes:

1. El voltaje "forward" (V_F): cuando los diodos están conectados en polarización directa hemos dicho que dejan pasar el flujo de la corriente libremente, pero ese comportamiento solamente aparece a partir de que el diodo es sometido a una determinada tensión "umbral". Es decir: mientras no haya entre los extremos de un diodo un voltaje superior a V_F , no pasará corriente eléctrica a través de él. Un diodo de silicio típico tiene un valor de V_F entre 0,6 y 1V. Los diodos de tipo Schottky tienen la característica de tener un V_F más bajo (entre 0,15V y 0,45V).

Lo más interesante de V_F , no obstante, reside en que su valor representa la caída de tensión permanente que existirá siempre entre los terminales del diodo, independientemente de la intensidad de corriente que circule por él (a la cual llamaremos I_F). Es decir, con los diodos no se cumple la Ley de Ohm. Por eso insistimos: sea cual sea la intensidad que atravesie el diodo (I_F la hemos llamado, recordemos) entre sus extremos aparecerá siempre el mismo valor V_F .

NOTA: En realidad, esto no es rigurosamente cierto porque sí que es verdad que un (gran) aumento de I_F conlleva un (leve) aumento en V_F pero este está delimitado a un rango de valores muy pequeño, así que, para simplificar, asimilaremos este rango de valores a un único valor V_F constante. Si el lector deseara conocer en detalle la relación I_F-V_F de un diodo en particular, recomiendo la consulta de su correspondiente datasheet.

Breve nota sobre los "datasheets"

Los "datasheets" son documentos redactados por los fabricantes de componentes electrónicos donde se exponen las diferentes características técnicas de un determinado producto (ya sea desde un simple diodo hasta un sensor, un motor o incluso un complejo microcontrolador). Allí podemos encontrar información sobre el uso y el rendimiento del componente en cuestión dentro de un circuito; por

ejemplo: la tensión/corriente mínima necesaria para funcionar o la máxima que puede soportar, la resistencia ofrecida dependiendo de las condiciones ambientales, la polaridad y funcionalidad de sus bornes/patillas, sus posibles modos de configuración, sus dimensiones, etc., etc. Es tremadamente útil y, tarde o temprano, deberemos consultarlos.

2. El voltaje "breakdown" (V_{BR}): Cuando los diodos están conectados en polarización inversa hemos dicho que no dejan pasar el flujo de la corriente (en realidad sí, la llamada "corriente de saturación inversa", pero es muy pequeña –del orden de los nA– y, por tanto, no la tendremos en cuenta) pero ese comportamiento es así solamente mientras al diodo no se le aplique un voltaje mayor del llamado "voltaje de ruptura". Es decir: si entre los terminales de un diodo conectado en polarización inversa hay una tensión mayor que V_{BR} , ese diodo permitirá el flujo de corriente sin ejercer oposición.

Un diodo de silicio típico tiene un valor de V_{BR} entre 50 y 100V (o incluso más); los diodos de tipo Schottky tienen un valor V_{BR} más bajo, entre 20 y 40V. En este sentido, los diodos de tipo Zener ofrecen valores de V_{BR} más bajos aún, y sobre todo muy estables y precisos, siendo útiles, por tanto, en situaciones donde se necesite generar una tensión conocida de referencia con ese valor (conectándose para ello en polaridad inversa respecto a la fuente de alimentación y en paralelo al circuito al que se desea ofrecer dicha tensión).

3. La corriente máxima ($I_{F(max)}$) que puede atravesar el diodo en polarización directa sin que este se funda (debido al calor generado por la potencia disipada, siendo $P_{maxsoportada}=V_F \cdot I_{F(max)}$).

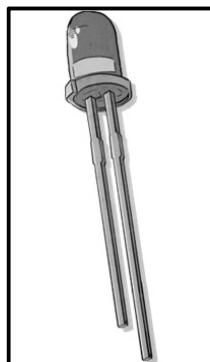
Para evitar superar en nuestros circuitos el valor concreto de $I_{F(max)}$ correspondiente al diodo usado, es muy recomendable conectar un divisor de tensión (es decir, una resistencia en serie) a uno de los terminales de ese diodo (no importa si es el ánodo o el cátodo). Para calcular el valor adecuado de esta resistencia, debemos tener en cuenta la intensidad que queremos que pase de forma segura por el diodo ($I_{FDeseada}$) –un diodo de silicio típico suele soportar intensidades de hasta 20-30 mA, aunque para una mayor exactitud, recomiendo la consulta de su datasheet–, la tensión existente entre los extremos del sistema diodo+resistor (V_{Sist}) –suponiendo que tenemos un circuito formado solamente por la conexión en serie de diodo y resistencia, esta tensión se correspondería con la total que aporta la fuente de alimentación– y el voltaje umbral del diodo (V_F); una vez conocido estos valores podremos calcular la mínima resistencia adecuada aplicando la Ley de Ohm y la 2^a Ley de Kirchhoff, así: $R=(V_{Sist}-V_F)/I_{FDeseada}$. Por otro lado, la potencia disipada por esa resistencia podríamos calcularla mediante la fórmula $P=(V_{Sist}-V_F) \cdot I_{FDeseada}$.

Un detalle que debemos tener en cuenta es que si conectamos varios diodos en serie (algo bastante habitual en el caso de utilizarlos como divisores de tensión, o bien si empleamos LEDs, que no son más que un tipo especial de diodo), los diversos V_F se suman (debido a la segunda Ley de Kirchhoff), así que para que ese circuito funcione (es decir, para que la corriente que lo atraviesa sea mayor de 0 pero menor de $I_{F(\max)}$), el valor de la resistencia a conectar en serie al resto de diodos vendrá dado por la expresión: $R = (V_{Sist} - n \cdot V_F) / I_{FDeseada}$, entendiendo "sistema" como el conjunto de diodos+resistencia en serie.

En el caso de querer conectar varios diodos entre sí en paralelo (también algo habitual en el caso de los LEDs) se ha de añadir una resistencia en serie a cada uno de ellos y tratar cada pareja resistencia+diodo como un sistema independiente (esto es: aplicar por separado a cada pareja la fórmula $R = (V_{Sist} - V_F) / I_{FDeseada}$). La razón de no usar una sola resistencia para todos los diodos en paralelo es porque sus respectivos valores de V_F pueden ser diferentes (esto pasa incluso entre diodos nominalmente iguales porque en su proceso de fabricación siempre se producen variaciones inevitables!) y, por tanto, el valor adecuado de R para un diodo puede no ser el adecuado para otro aun siendo, insistimos, aparentemente idénticos.

Los diodos de la familia 1N400x (todos de silicio) son los más comunes en los circuitos similares a los mostrados en este libro; todos ellos funcionan perfectamente a una I_F de 1A y tienen un V_F aproximado de 0,7V; lo que los diferencia es su respectivo V_{BR} (el modelo 1N4001, por ejemplo, tiene un V_{BR} de 50V, el 1N4002 de 100V, etc.). Concretamente, Adafruit distribuye el diodo 1N4001 como producto **nº 755** y Sparkfun como producto **nº 8589**. Este último también distribuye un diodo de tipo Schottky (con I_F óptima de 1A y V_{BR} de 40V) como producto **nº 10926** y un diodo de tipo Zener (con un V_{BR} de 5,1V) como producto **nº 10301**. DFRobot, por su parte, ofrece como producto **FIT0323** un kit con varios diodos de características diferentes.

Un tipo de diodo muy particular: el LED



Un "Light Emitting Diode" (LED) es, como su nombre indica, un diodo que tiene una característica peculiar: emite luz cuando la corriente eléctrica lo atraviesa. De hecho, lo hace de forma proporcional: a más intensidad de corriente recibida (y por tanto, a más potencia consumida), más luz radiada. Esto significa que modificar el brillo de un LED es tan sencillo como modificar la intensidad de corriente que fluye por él.

Ya que no deja de ser un tipo concreto de diodo, también puede ser conectado en polarización directa o inversa, teniendo en

EL MUNDO GENUINO-ARDUINO

cuenta que solo se iluminarán si están conectados en polarización directa. Por ello, cuando diseñemos nuestros circuitos hay que seguir teniendo la precaución de conectar cada terminal del LED en la polaridad adecuada. No obstante, como a un LED no se le puede pintar una marca encima, la manera de distinguir el ánodo ("terminal positivo" en polarización directa) del cátodo (el "terminal negativo" en polarización directa) es observando su longitud: el ánodo es de una longitud más larga que el cátodo. Si, por alguna razón, ambos tuvieran la misma longitud, aún se puede saber cuál es el cátodo identificando el terminal situado bajo la parte plana de la carcasa luminosa del LED.

Igualmente, como un LED no deja de ser un tipo concreto de diodo, sigue siendo muy recomendable conectarle una resistencia en serie para limitar la intensidad de corriente que lo atraviesa (y así mantenerla por debajo del valor máximo más allá del cual el LED puede dañarse, es decir, $I_{F(\max)}$). Para calcular qué resistencia debemos colocar, podemos utilizar la misma fórmula mencionada en párrafos anteriores: $R = (V_{Sist} - V_F) / I_{FDeseada}$, sabiendo que, de forma general, la intensidad que suele venir bien para el funcionamiento óptimo de un LED típico (es decir, $I_{FDeseada}$) es de unos 15-20mA y que la tensión V_F apropiada varía según el color del LED: va de 3V a 3,6V para el ultravioleta (UV), blanco o azul, de 2,5V a 3V para el verde, de 1,9V a 2,4V para el rojo, naranja, amarillo o ámbar y de 1V a 1,5 para el infrarrojo. En cualquier caso, para mayor seguridad es recomendable consultar siempre las especificaciones que proporciona el fabricante (el "datasheet" del componente) para conocer toda la información necesaria sobre intensidades y tensiones soportadas.

Además de por sus diferentes colores (consecuencia del material de fabricación usado, diferente para cada tipo de LED), podemos clasificar estos componentes según si emiten la luz de forma difusa o clara. Los primeros se suelen utilizar para indicar presencia, ya que emiten una luz suave y uniforme que no deslumbra y que puede verse bien desde cualquier ángulo. Los segundos sirven para irradiar en una dirección muy concreta con luz directa y potente, por lo que no se ven bien en todos los ángulos pero iluminan mucho más que los otros. En cualquier caso, sea un LED de tipo difuso o claro, para saber en términos cuantitativos lo "brillante" que es su luz, será necesario conocer la cantidad de milicandelas (mcd) que ese LED concreto es capaz de emitir; este dato (conocido como "intensidad luminosa") lo debe ofrecer el fabricante en el datasheet.

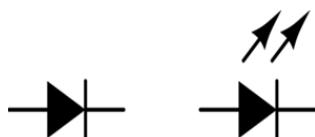
La mayoría de distribuidores de componentes electrónicos ofrecen la posibilidad de adquirir además de LEDs individuales de varios colores, paquetes ya preparados que incluyen diversas cantidades de LEDs diferentes. Un ejemplo de esto

es el producto nº **9881** de Sparkfun, el cual contiene tanto LEDs difusos –ahí llamados "Basic"– y claros –ahí llamados "Super Bright"– (todos ellos de 5mm de diámetro) o el producto nº **11459**, el cual también contiene LEDs difusos y claros (pero en este caso de 10mm de diámetro). Otros paquetes interesantes ofrecidos por Sparkfun son el producto nº **12062** y el nº **12903**, los cuales contienen LEDs solamente difusos de 5mm de colores variados. DFRobot distribuye, por su parte, un kit de LEDs claros (de 3mm de diámetro y de diferentes colores) como producto **FIT0244**, otro kit de LEDs claros (de 5mm de diámetro y también de diferentes colores) como producto **FIT0242** y otro kit de LEDs (esta vez difusos de 5mm y diferentes colores) como producto **FIT0372**. Adafruit no distribuye paquetes con LEDs de diferentes colores, pero sí ofrece paquetes con LEDs de un determinado color: por ejemplo, su producto nº **777** corresponde a un paquete con LEDs difusos de 3mm de color rojo, el nº **299** a otro con LEDs difusos de 5 mm (también de color rojo), el nº **845** a otro con LEDs difusos de 10mm (también de color rojo), el nº **297** a otro con LEDs claros de 5mm (también de color rojo), etc.

Breve nota sobre los LEDs RGB

Si el lector se entretuviera en consultar el catálogo de cualquier distribuidor electrónico, vería que la variedad de colores existente para el surtido de LEDs disponibles no es muy elevada: las opciones para elegir son solamente blanco, amarillo, naranja, rojo, verde, azul, violeta y poco más. Para conseguir emitir luz de un color diferente a estos dados, una posibilidad es utilizar los llamados LEDs RGB, cuyo funcionamiento se detallará en el capítulo 6 (ya que para sacarles provecho necesitamos conocer el funcionamiento de la placa Arduino). El nombre de RGB proviene de "Red-Green-Blue", los tres colores a partir de los cuales, dentro del ámbito de la electrónica, se pueden generar el resto de colores (es decir, los llamados tres colores "primarios").

El símbolo que se suele utilizar en el diseño de circuitos electrónicos para representar un diodo estándar es el mostrado a la izquierda en la imagen siguiente, y el de un LED es el de la derecha. El vértice del triángulo secante con la línea perpendicular de ambos símbolos representa el cátodo. También existen otros símbolos similares que representan diodos más específicos (como los de tipo Zener o Schottky, entre otros).



Condensadores

El condensador es un componente cuya función básica es almacenar carga eléctrica en cantidades limitadas, de manera que esta se pueda utilizar a modo de "fuente de alimentación alternativa" en ocasiones puntuales (concretamente, cada vez que se cree un circuito cerrado entre sus terminales sin presencia de otras fuentes DC); por otro lado, cuando un condensador está completamente cargado, en circuitos de corriente DC actúa como un interruptor abierto. En otras palabras: cuando una corriente eléctrica (proveniente de una fuente externa) llega hasta el condensador, este se empieza a cargar. A medida que esto ocurre, el condensador va impidiendo cada vez más el paso de electrones a través de sí mismo, hasta que una vez ya está cargado completamente, deja definitivamente de conducir la corriente "hacia el otro lado". En el momento que la corriente deje de llegar al condensador (esto es, cuando se desconecte la fuente externa), este empezará a descargarse, liberando al circuito la carga eléctrica que tenía almacenada y, por tanto, generando durante unos instantes una nueva corriente eléctrica, hasta quedar completamente descargado. El extremo del condensador que durante su descarga actúa como "borne positivo" es siempre el que está conectado al borne positivo de la fuente externa y el extremo que actúa como "borne negativo" es siempre el que está conectado a tierra.

La capacidad (C) de un condensador es su característica más importante y se puede definir como la relación –normalmente de un valor constante– que existe entre la cantidad de carga eléctrica (Q) que almacena en un momento determinado y el voltaje (V) que se le está aplicando en ese mismo momento. Concretamente, se define así: $C=Q/V$.

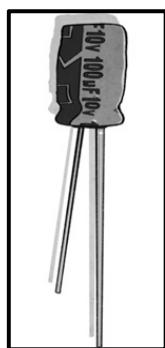
De la fórmula anterior podemos deducir varias cosas: la primera es que un condensador con mayor capacidad que otro almacenará más carga bajo el mismo potencial. La segunda es que un condensador con una determinada capacidad almacenará más carga cuanto mayor sea el voltaje aplicado (aunque en este sentido hay que tener en cuenta que todo condensador tiene un voltaje de trabajo máximo –que suele venir impreso en el cuerpo del propio condensador– más allá del cual se puede dañar, por lo que siempre se tendrá un máximo de carga almacenable).

También es fácil deducir que si se conecta una resistencia en serie a un condensador, se estará aumentando el tiempo que tarda este en cargarse y descargarse (de hecho, cuanto mayor sea el valor de dicha resistencia, mayor será ese tiempo); esto es porque al añadir una resistencia en el camino de los electrones, se está reduciendo la intensidad de corriente que pasa por ahí y, por tanto, se está aumentando el tiempo necesario para que llegue/salga una determinada cantidad de carga al/del condensador.

La capacidad se mide en faradios (F), aunque la mayoría de condensadores con los que trabajaremos tienen una capacidad mucho menor (del orden de los microfaradios nanofaradios o incluso picofaradios). Dependiendo del tamaño del condensador, puede ser que el valor de su capacidad no pueda ser serigrafiado tal cual sobre su cuerpo; en esos casos, se suele utilizar una secuencia de tres dígitos para indicar las dos primeras cifras del valor de la capacidad y luego su multiplicador (tomando como unidad el picofaradio). Por ejemplo, un condensador con el número "403" impreso, querrá decir que tiene una capacidad de $40 \cdot 10^3 = 40 \cdot 1000 = 40000 \text{ pF} = 40 \text{ nanofaradios}$.

Al igual que ocurría con las resistencias, los condensadores pueden ser conectados en serie o en paralelo para conseguir un circuito con una capacidad equivalente. En concreto, es posible demostrar que si dos o más condensadores ($C_1, C_2, C_3\dots$) se conectan en serie, la capacidad equivalente se puede calcular mediante la fórmula $1/C=1/C_1+1/C_2+1/C_3+\dots$ (resultando, por tanto, siempre menor que cualquiera de la de los condensadores individuales). Si los condensadores se conectan en paralelo, la capacidad equivalente es $C=C_1+C_2+C_3+\dots$ (es decir, obtenemos una capacidad total mayor). En cualquier caso, para obtener unos resultados ajustados a la realidad, al realizar estos cálculos hay que tener en cuenta el % de tolerancia señalado por el fabricante de los condensadores, valor que suele indicarse mediante un código de letras estandarizado ($F=\pm 1\%$, $G=\pm 2\%$, $J=\pm 5\%$, $K=\pm 10\%\dots$ y así).

También es necesario tener presente tres características importantes en cualquier condensador: 1^{a)} que, en la práctica, todo condensador ofrece una resistencia intrínseca al paso de corriente (aunque es muy pequeña, del orden de $0,01\Omega$ o menos); 2^{a)} que, estando cargados, todos ellos sufren leves fugas de electrones (del orden de $1nA$ o menos) que provocan una lentísima (pero existente) descarga paulatina y 3^{a)} que tienen una tensión de trabajo máxima por encima de la cual pueden fundirse y/o quemarse.

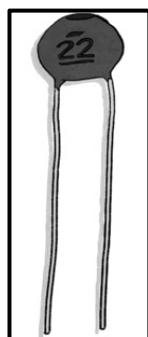


Podemos clasificar los tipos de condensadores según si tienen o no polarización. Los condensadores polarizados son los que se han de conectar al circuito respetando el sentido de la corriente. Es decir, tienen un terminal "negativo" (cátodo) que siempre deberá conectarse al polo negativo del circuito, y otro terminal "positivo" (ánodo) que siempre deberá conectarse al polo positivo. Dicho de otra forma: su conexión se ha de realizar siempre en polarización directa. Es por ello que este tipo de condensadores no son válidos para ser usados en corriente alterna; de hecho, al conectarlos en polarización inversa son destruidos. Para distinguir un terminal del

EL MUNDO GENUINO-ARDUINO

otro, podemos fijarnos en una franja pintada sobre el cuerpo del condensador, la cual indicará siempre el cátodo (de forma similar a lo que ocurría con los diodos). Otra pista que podemos usar para lo mismo es fijarnos en que el cátodo es más corto que el ánodo (al igual que ocurría con los LEDs). Como características más reseñables, podemos decir que los condensadores polarizados tienen por norma una capacidad bastante elevada (de un valor mayor de 1 microfaradio), y, teniendo en cuenta el modo de fabricación, suelen ser de tipo electrolítico, bien de aluminio (como el mostrado en la imagen) o bien de tantalio, principalmente.

Sparkfun distribuye un condensador electrolítico de $10\mu F$ y otro de $100\mu F$ (ambos con un voltaje de trabajo máximo de 25V) como productos **nº 523** y **nº 96**, respectivamente. Adafruit distribuye un condensador electrolítico de $10\mu F$ (con voltaje máximo de 50V) como producto **nº 2195**, otro de $47\mu F/25V$ como **nº 2194**, otro de $100\mu F/16V$ como **nº 2193**, otro de $220\mu F/16V$ como **nº 2192** y otro de $4700\mu F/10V$ como **nº 1589**. DFRobot, por su parte, además de distribuir un supercondensador electrolítico de $100F/2,7V$ como producto **FIT0134** y otro de $4,7F/2,7V$ como producto **FIT0133**, también ofrece un kit con múltiples condensadores electrolíticos diferentes (con capacidades $1\mu F$ entre y $470\mu F$) como producto con código **FIT0117**.

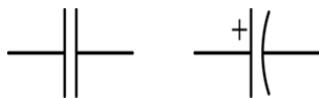


Los condensadores unipolares (es decir, no polarizados) pueden conectarse al circuito en ambos sentidos indiferentemente (al igual que las resistencias, por ejemplo). Pueden estar fabricados de muchos materiales, pero los más comunes son los de tipo plástico o cerámico (como el mostrado en la imagen). En general, suelen tener una capacidad bastante menor que los condensadores polarizados, por lo que su tiempo de carga (y descarga) total también suele ser menor.

Sparkfun distribuye un condensador cerámico de $22pF$ (con un voltaje de trabajo máximo de 200V) y otro de $0,1\mu F$ (con un voltaje de trabajo máximo de 50V) como productos **nº 8571** y **nº 8375**, respectivamente. Adafruit distribuye solamente este último como producto **nº 753**. DFRobot distribuye un kit con múltiples condensadores cerámicos diferentes como producto con código **FIT0118**. Finalmente, Sparkfun también distribuye un kit de múltiples condensadores diferentes (pero esta vez tanto electrolíticos como cerámicos) con número de producto **nº 13698**.

Los símbolos utilizados en el diseño de los circuitos eléctricos para representar un condensador pueden ser dos: la representación de la izquierda es la de un condensador unipolar, y la de la derecha es la de un condensador polarizado.

En este último, la línea recta simboliza el ánodo (a veces el signo "+" se omite y/o la línea recta se pinta más gruesa) y la línea curva simboliza el cátodo:



Usos comunes de los condensadores: desacople y filtro

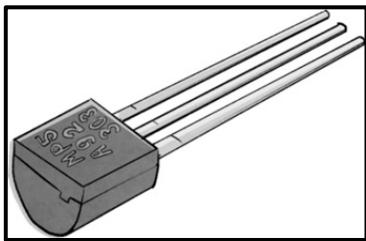
Los condensadores muchas veces se utilizan en los circuitos para proporcionar la llamada "alimentación de desvío" o "desacople" (en inglés, "by-pass" o "decoupling"). Esta alimentación es necesaria cuando un componente que normalmente no requiere de mucha intensidad de corriente para funcionar, ha de realizar de forma puntual un consumo tan elevado de electricidad que la fuente de alimentación ordinaria no es capaz de ofrecerla con la celeridad suficiente. Esto normalmente ocurre cuando dicho componente (un microcontrolador como el que incorpora la placa Arduino, por ejemplo) pasa de estar en estado desactivado a activado; en ese momento es cuando el condensador proporciona rápidamente la corriente transitoria necesaria al "soltar" la carga eléctrica que mantenía almacenada. Con este sistema de alimentación "alternativa", se logra dar una rápida respuesta al pico de consumo del componente mientras que la fuente de alimentación puede ir volviendo a cargar otra vez el condensador (a un ritmo menor) de cara al próximo pico. Para utilizar un condensador que realice esta función, uno de sus terminales ha de conectarse lo más cerca posible de la entrada de alimentación del componente a "gestionar" (cuanto más lejos, menor es el efecto) y el otro terminal a la tierra del circuito. Valores típicos de un condensador "by-pass" son $0,1\ \mu F$ o $0,01\ \mu F$.

Otro uso muy frecuente de los condensadores es la eliminación del "ruido" de la señal de alimentación DC (sobre todo si viene originada por un rectificador). Es decir, aunque una fuente sea etiquetada nominalmente como de 9V, por ejemplo, en la realidad nunca ofrecerá esos 9V exactos, sino que ese valor irá sufriendo variaciones más o menos amplias alrededor de su valor nominal. Por tanto, además de una alimentación DC tenemos siempre una pequeña señal AC alrededor de aquella. Dependiendo de la magnitud de esta señal AC (es decir, de las variaciones alrededor de los 9V del ejemplo), podríamos llegar a tener un problema en nuestro circuito, porque aparecerían efectos de corrientes de tipo AC que no deseamos. Un condensador es capaz de estabilizar esas variaciones, permitiendo obtener de la fuente un valor de tensión más constante, gracias a que es capaz de regular convenientemente la carga que "suelta" o "acumula" en función del voltaje fluctuante al que es sometido. Para utilizar un condensador que realice esta función de estabilización de la señal (los cuales se suelen llamar "condensadores de filtro"), uno

EL MUNDO GENUINO-ARDUINO

de sus terminales se ha de conectar al borne positivo de la fuente de alimentación y el otro terminal se ha de conectar al borne negativo (a tierra). Un valor típico de un condensador de filtro es 0,1 μF , aunque para frecuencias elevadas de corriente AC se necesitarán condensadores (o equivalencias de ellos conectados en serie y/o paralelo) de menor capacidad (para que el ciclo carga-descarga sea más rápido).

Transistores



Un transistor es un dispositivo electrónico que restringe o permite el flujo de corriente eléctrica entre dos contactos según la presencia o ausencia de corriente en un tercero. Puede entenderse como una resistencia variable entre dos puntos, cuyo valor es controlado mediante la aplicación de una determinada corriente sobre un tercer punto.

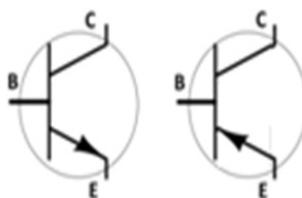
Los transistores se suelen utilizar como amplificadores de corriente, ya que con una pequeña corriente recibida a través de su terminal de control permiten la circulación de una intensidad muy grande (proporcional a aquella, hasta un máximo) entre sus dos terminales de salida. Otro uso muy frecuente es el de ser interruptores, ya que si su terminal de control no recibe ninguna intensidad de corriente, por entre los dos terminales de salida no fluye ninguna corriente tampoco, abriendo el circuito.

Existen dos grandes categorías de transistores según su tecnología de fabricación y funcionamiento: los transistores de tipo bipolar (llamados comúnmente BJT, del inglés "Bipolar Junction Transistor") y los transistores de tipo efecto de campo (llamados comúnmente FET, del inglés "Field Effect Transistor"). A continuación estudiaremos los primeros y después comentaremos brevemente los segundos.

El encapsulado físico de los transistores BJT puede ser muy diverso, pero lo más habitual es que estos componentes dispongan de tres patillas (tal como muestra la figura anterior), las cuales tienen, cada una de ellas, un nombre específico: "Colector", "Base" y "Emisor". Para saber, no obstante, qué patilla de las tres es la "Base", cual es el "Colector" y cual el "Emisor", será necesario consultar el datasheet del modelo concreto de transistor utilizado. Sea cual sea, la "Base" siempre hace de "terminal de control" y el "Colector" y el "Emisor" siempre son los "terminales de salida".

Podemos clasificar a su vez los transistores BJT en dos tipos, los NPN -los más habituales, con diferencia- y los PNP. En el caso de los NPN, si aplicamos cierta

corriente (por lo general muy baja) de la Base al Emisor, el Emisor actuará como una válvula que regulará el paso de corriente desde el Colector hacia el propio Emisor. En el caso de los PNP, si aplicamos cierta corriente (por lo general muy baja) del Emisor a la Base, el Emisor actuará como una "válvula" que regulará el paso de corriente desde el propio Emisor hacia el Colector. Los símbolos utilizados en el diseño de los circuitos eléctricos para representar los transistores NPN –izquierda– y PNP –derecha– intentan, de hecho, ilustrar gráficamente este comportamiento:



Cuando un transistor BJT se utiliza como amplificador, la intensidad de corriente que circula del Emisor al Colector (si es PNP) o del Colector al Emisor (si es NPN) puede llegar a ser decenas de veces mayor que la corriente existente en el terminal de entrada Base-Emisor. Este factor de proporcionalidad entre la intensidad que pasa por la Base y la que pasa por el Colector (es decir, el factor de amplificación o ganancia) depende del tipo de transistor y ha de venir descrito en sus especificaciones técnicas (el "datasheet") con el nombre de β o h_{FE} . Más en concreto, podemos distinguir tres "modos de funcionamiento" en un transistor NPN típico:

El modo de corte: se produce cuando la corriente que fluye por la Base (I_b) es próxima a 0 (o, más en concreto, mientras la diferencia de tensión entre la Base y el Emisor (V_{eb}) sea –generalmente– menor de 0,7V, valor-umbral que podríamos considerar análogo al V_F visto en los diodos). En ese caso, no existe paso de corriente ni por el Emisor ni por el Colector (es decir, $I_c=I_e=0$), por lo que el transistor se comporta como un interruptor abierto.

El modo de saturación: se produce cuando la corriente que fluye por el Colector (I_c) es prácticamente idéntica a la que fluye por el Emisor (I_e), momento en el cual, de hecho, ambas se aproximan al valor máximo de corriente que puede soportar el transistor en sí. Es decir, el transistor se comporta como una simple unión de cables, ya que la diferencia de potencial entre Colector y Emisor en este modo (la llamada $V_{ce(sat)}$) es muy próxima a cero (típicamente, entre 0,05V-0,2V). Este modo se da cuando la intensidad que circula por la Base supera un cierto valor umbral ($I_{b(sat)}$).

EL MUNDO GENUINO-ARDUINO

El modo activo: se produce cuando el transistor no está ni en su modo de corte ni en su modo de saturación (es decir, en un modo intermedio). Es en este modo cuando la corriente de circula por el Colector depende de forma directamente proporcional de la corriente que circula por la Base (mayor que cero pero menor que $I_{b(sat)}$) y de β (la ganancia de corriente), esto es: $I_c = \beta \cdot I_b$. También se cumple que $I_e = I_b + I_c$ (de lo que se deduce que $I_c = \alpha \cdot I_e$, —donde α es igual a $\beta / (\beta + 1)$ — y, por tanto, tendrá siempre un valor muy cercano a 1 pero menor—) y que $V_{ce} > V_{ce(sat)}$.

Como se puede ver, el modo activo es el interesante para usar el transistor como un amplificador lineal de señal ($I_c = \beta \cdot I_b$), y los modos de corte y saturación son los interesantes para usar el transistor como un interruptor entre colector y emisor.

A la hora de adquirir un transistor BJT (nos centraremos en los de tipo NPN por ser los más utilizados en circuitos similares a los mostrados en este libro) es importante tener en cuenta dos datos más: uno es $V_{ce(max)}$, que representa la caída de tensión máxima que se puede aplicar entre el colector y el emisor más allá de la cual el transistor se estropeará, y el otro es $I_{c(max)}$, que representa la intensidad de corriente máxima que puede fluir por el colector más allá de la cual el transistor también se estropeará. Ligado a estos dos datos está además un tercer dato importante: la potencia máxima (en W) que el transistor puede disipar. Finalmente, un detalle que también deberíamos tener en cuenta es que el valor de la ganancia de un transistor (β , recordemos) suele ser diferente según el V_{ce} y la I_c existentes en cada momento.

Sparkfun distribuye el transistor NPN 2N3904 como producto **nº 521** (el cual admite 40V de $V_{ce(max)}$ y 200mA de $I_{c(max)}$), el BC547 como producto **nº 8928** (el cual admite 45V de $V_{ce(max)}$ y 100mA de $I_{c(max)}$) y el BC337 como producto **nº 13689** (el cual admite 50V de $V_{ce(max)}$ y 800mA de $I_{c(max)}$). DFRobot distribuye con código **FIT0322** un kit con varios modelos de transistores NPN —y también PNP— de diferentes características. Adafruit solamente distribuye el transistor NPN PN2222A como producto **nº 756** (el cual admite 40V de $V_{ce(max)}$ y 1A de $I_{c(max)}$).

Todos los transistores mencionados en el párrafo anterior ofrecen un valor de ganancia similar, de alrededor de 100 puntos en condiciones estándares. Si quisieramos conseguir más ganancia aún, podríamos utilizar entonces lo que se llama un transistor Darlington. Este tipo de transistor está compuesto internamente por dos transistores (normalmente NPN) conectados en cascada (esto significa que el primer transistor entrega la corriente que sale por su emisor a la base del segundo transistor)

CAPÍTULO 1: ELECTRÓNICA BÁSICA

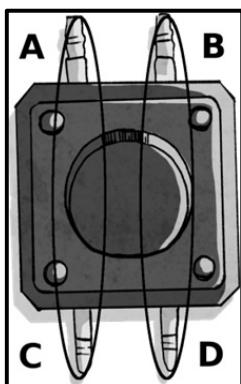
y su alta ganancia proviene de la multiplicación de la ganancia de sus dos transistores. Adafruit distribuye el Darlington TIP120 como producto **nº 976** (el cual admite 60V de $V_{ce(max)}$ y 5A de $I_{c(max)}$ con una ganancia de 1000).

Por su lado, los transistores FET cumplen la misma función que los BJT (amplificador o interruptor de corriente, entre otras), pero sus tres terminales se denominan (en vez de Base, Emisor y Colector): Puerta (identificado como "G", del inglés "Gate"), Surtidor (S) y Drenador (D). El terminal G sería el "equivalente" a la Base en los BJT, pero la diferencia está en que el terminal G no absorbe corriente en absoluto (frente a los BJT donde la corriente atraviesa la Base pese a ser pequeña en comparación con la que circula por los otros terminales). El terminal G más bien actúa como un interruptor controlado por tensión, ya que (y aquí está la clave del funcionamiento de este tipo de transistores) será el voltaje existente entre G y S lo que permita que fluya o no corriente entre S y D. De hecho, el comportamiento de un transistor FET también se puede dividir en tres regiones de operación diferentes (llamadas "de corte", "de saturación" y "lineal") pero ahora dependiendo de las tensiones existentes entre sus terminales.

Así como los transistores BJT se dividen en NPN y PNP, los transistores FET son también de dos tipos: los de "canal n" y los de "canal p", dependiendo de si la aplicación de una tensión positiva en la puerta pone al transistor en estado de conducción o no conducción, respectivamente. Por otro lado, los transistores FET también se pueden clasificar a su vez dependiendo de su estructura y composición interna. Así tenemos los transistores JFET (Junction FET), los MOS-FET (Metal-Oxide-Semiconductor FET) o MIS-FET (Metal-Insulator-Semiconductor FET), entre otros. Cada uno de estos tipos tiene diferentes características específicas que los harán más o menos interesantes dependiendo de las necesidades del circuito, y cada uno tiene un símbolo esquemático diferente.

En general, los transistores FET se suelen utilizar más que los BJT en circuitos que consumen gran cantidad de potencia y/o cuando se necesita pasar del estado de corte al de saturación (o viceversa) mucho más rápidamente. Por ejemplo, el transistor MOS-FET de "canal n" FQP30N06L, distribuido por Sparkfun como producto **nº 10213**, admite hasta 60V entre D y S y hasta 30A circulando por D (estos dos datos son ahora los factores limitantes); el transistor FQP27P06 (de "canal p") distribuido por Adafruit como producto **nº 1794** admite igualmente 60V_{DS} y hasta 27A_D; el transistor IRLB8721 (de "canal n") distribuido por Adafruit como producto **nº 355**, en cambio, admite "solo" 30V_{DS} como máximo pero soporta hasta 60A_D.

Pulsadores



Ya sabemos que un interruptor es un dispositivo con dos posiciones físicas: en la posición de "cerrado" se produce la conexión de dos terminales (lo que permite fluir a la corriente a través de él) y en la posición de "abierto" se produce la desconexión de estos dos terminales (y por tanto se corta el flujo de corriente a través de él). En definitiva, que un interruptor no es más que un mecanismo constituido por un par de contactos eléctricos que se unen o separan por medios mecánicos.

Un pulsador (en inglés, "pushbutton") es un tipo de interruptor en el cual se establece la posición de encendido mediante la pulsación de un botón gracias a la presión que se ejerce sobre una lámina conductora interna. En el momento de cesar la pulsación sobre dicho botón, un muelle hace recobrar a la lámina su posición primitiva, volviendo a la posición de "abierto".

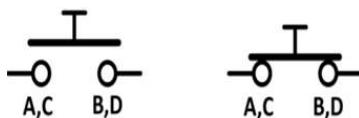
Existe una gran variedad de pulsadores de muchas formas y tamaños diferentes, pero en los circuitos que realizaremos a lo largo de este libro utilizaremos unos pulsadores muy prácticos y relativamente pequeños (con un tamaño de tan solo 6x6mm) que se pueden adquirir en cualquier distribuidor listado en el apéndice A (por ejemplo, en DFRobot es el producto **FIT0179**, en Sparkfun el **nº 97** y en Adafruit el **nº 367** –también podemos considerar el **nº 1490**–) y que son los que habitualmente vienen en los kits de aprendizaje. Otro pulsador similar también distribuido por ambas empresas (pero con un tamaño de 12x12mm) es el producto **nº 9190/1119** (Sparkfun/Adafruit, respectivamente). Asimismo podemos adquirir un surtido de varios pulsadores de colores (también de 12mm) como productos **nº 10302/1009**.

Todos estos pulsadores tienen una característica que conviene aclarar: tal como se ve en la imagen anterior, tienen cuatro patillas. Ello nos puede hacer pensar que tienen cuatro terminales, pero nada más lejos de la realidad: las dos patillas enfrentadas de cada lado están unidas internamente entre sí, por lo que funcionan como una sola. Es decir, las patillas A y C resaltadas en la imagen representan un solo punto eléctrico, y las patillas B y D representan otro punto eléctrico único. Por tanto, realmente, este pulsador solamente tiene dos terminales: A/C y B/D.

Hay que tener en cuenta, por otro lado, que este tipo de pulsadores tienen una cantidad máxima bastante limitada de corriente que pueden resistir antes de quemarse (suele ser generalmente un valor alrededor de 50mA), así que si se va a utilizar una alimentación más exigente, se deberán utilizar pulsadores más robustos.

En cualquier caso, para conocer los valores concretos de tensión, corriente (y presión sobre el botón, entre otras cosas) máximos que puede soportar un pulsador concreto, nos hemos de remitir a la documentación oficial que proporciona el fabricante para ese componente (el "datasheet").

En la imagen siguiente se puede ver el símbolo eléctrico del pulsador (a la izquierda en estado abierto y a la derecha en estado cerrado), indicando además como ejemplo a qué patillas físicas corresponde cada terminal:



Otros tipos de interruptores (o conmutadores)

Además de los pulsadores, existen muchos otros tipos de interruptores: basculantes, giratorios, de membrana, etc., etc. Cada uno de estos tipos de interruptores tiene un conjunto de características únicas que lo diferencian de los demás, tales como la acción que activa el interruptor, el periodo de mantenimiento de dicha activación, cuántos circuitos del interruptor puede controlar o el modo de conexión al resto del circuito, entre otras. Así pues, podemos clasificar un interruptor de las siguientes formas:

***Según el tipo de acción física** a realizar para cambiar el estado del interruptor (de abierto a cerrado, o viceversa). Esta acción puede ser mecánica (un empuje/tiro, un deslizamiento, un balanceo, un giro...) pero también magnética, térmica, etc. Cualquier interacción física que provoque una unión (o desunión) entre los contactos internos del interruptor puede ser utilizada.

***Según la permanencia de la unión** entre los contactos internos del interruptor, este puede ser clasificado como "momentáneo" o "mantenido". Los del primer tipo solamente se mantienen activos mientras están siendo utilizados y dejan de mantenerse así cuando dejan de serlo; los pulsadores mencionados en el apartado anterior son de tipo momentáneo; otro ejemplo pueden ser las teclas de un computador. Los del segundo tipo (también llamados "toggle" o "On/Off") se activan con una sola interacción, manteniéndose activados permanentemente sin necesidad de realizar ninguna otra acción, y así siguen hasta que vuelven a recibir una nueva

EL MUNDO GENUINO-ARDUINO

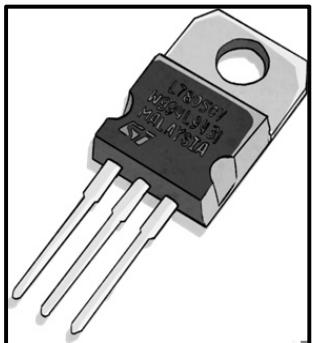
interacción que los desactiva; los interruptores domésticos de la luz son un ejemplo, y algunos pulsadores concretos también lo son.

Cuando un interruptor momentáneo no es utilizado, decimos que está en estado desactivado o "normal". Dependiendo de cómo fue construido ese interruptor, su estado normal puede provocar que el circuito esté abierto o cerrado. Si es el primer caso (es decir, si debemos usarlo para cerrar el circuito), el fabricante ha de indicar que se trata de un interruptor "normalmente abierto" (o NO, de "normally open"); si es el segundo caso (es decir, si debemos usarlo para abrir el circuito), estaremos hablando de un interruptor "normalmente cerrado" (o NC, de "normally closed"). Los pulsadores que utilizaremos en este libro son de tipo NO (los más habituales).

***Según la cantidad de circuitos que puede controlar.** Los pulsadores mencionados en el apartado anterior son la versión más simple que puede haber de un interruptor: tienen un terminal por donde la corriente entra y otro por donde (eventualmente) sale. Técnicamente esto es lo que se llama un interruptor de tipo SPST (es decir, "single-pole, single-throw", donde el número de "poles" –polos– define el número de circuitos que pueden ser controlados con el interruptor, y el número de "throws" –tiros– define el número de terminales de salida al que cada polo se puede conectar. Otros interruptores comunes son los de tipo SPDT ("single-pole, dual-throw"), los cuales han de tener (al menos) tres patillas físicas: una correspondiente al polo común y dos a cada uno de los tiros. Este tipo de interruptores (también llamados "comutadores", porque de hecho su tarea es esa) son muy utilizados, por ejemplo, para seleccionar la fuente de alimentación de nuestro circuito entre dos posibles fuentes disponibles. Añadiendo un polo a un interruptor SPDT obtenemos uno de tipo DPDT ("dual-pole, dual-throw"), los cuales deberían tener seis patillas físicas (2 polos y 2 tiros por cada polo, es decir, 4 en total), y así.

Listar aquí la extensísima variedad de conmutadores disponibles en Adafruit o Sparkfun (por no mencionar otros distribuidores online) no tiene mucho sentido porque cada uno de ellos es adecuado para unas necesidades muy concretas. Si el lector quiere conocer, de todas formas, qué es lo que ofrecen estas empresas, se puede consultar la lista completa accediendo a la categoría "Switches" de sus respectivas tiendas online; concretamente, aquí: (<https://www.sparkfun.com/categories/145>), o aquí: (<http://www.adafruit.com/categories/235>).

Reguladores de tensión



Un regulador de tensión es un componente electrónico que protege partes de un circuito (o un circuito entero) de elevados voltajes o de variaciones pronunciadas de este. Su función es proporcionar, a partir de un voltaje recibido fluctuante dentro de un determinado rango (el llamado "voltaje de entrada"), otro voltaje (el llamado "voltaje de salida") regulado a un valor estable y menor. Esto lo consigue aplicando la Ley de Ohm ($V=I \cdot R$): gracias a su capacidad de elevar o disminuir su resistencia interna, puede transferir sin perturbaciones

la intensidad de corriente ofrecida por la fuente al circuito (la cual atraviesa el propio regulador y varía según el consumo que el circuito necesite realizar en cada momento) manteniendo siempre a un valor constante –y reducido– el voltaje aplicado a ese circuito. Es decir: si para un determinado V de salida el circuito demandara más I , el regulador se lo ofrecerá disminuyendo su R interna; si demandara menos I , se lo ofrecerá igualmente, aumentando su R para mantener siempre el mismo V de salida. Así explicado podría parecer un simple divisor de tensión con resistencia variable (y de hecho, su función es la misma), pero su mecanismo de regulación del voltaje de salida es mucho más sofisticado y fiable.

Los reguladores de tensión son un elemento clave para conseguir una alimentación correcta y segura de los distintos componentes electrónicos de nuestros circuitos. Gracias a ellos, componentes que resultarían dañados si son sometidos a un voltaje demasiado elevado, pueden ser combinados en un mismo circuito por otros componentes más capaces y que requieran de una tensión mayor.

Existen muchos tipos de reguladores, pero en nuestros circuitos normalmente utilizaremos pequeños componentes llamados genéricamente reguladores LDO (del inglés "low-dropout"). El voltaje "dropout" es la diferencia entre el voltaje de entrada y el de salida. Este voltaje multiplicado por la corriente que lo atraviesa (es decir, la consumida en ese momento por el circuito ejerciendo de "carga") es gastado en forma de calor ($P=V_{\text{dropout}} \cdot I$), por lo que cuanto menor sea V_{dropout} , más eficiente será el regulador en términos de pérdida de energía (así como cuanto menos corriente consuma el circuito). En este sentido, todos los reguladores tienen un máximo de intensidad soportada más allá del cual dejan de funcionar irreversiblemente, por lo que es muy importante elegir el modelo de regulador adecuado dependiendo del consumo esperado del circuito a alimentar. Así pues, un regulador LDO alimentado a 12V que ofrezca una tensión regulada de 5V y soporte

EL MUNDO GENUINO-ARDUINO

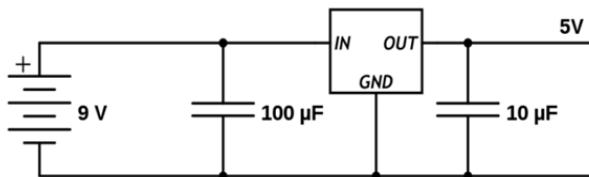
como mucho una intensidad de corriente de 1,2A sin dañarse (un caso bastante típico) perderá en forma de calor como máximo la siguiente potencia: $P=V \cdot I = (12V - 5V) \cdot 1,2A = 8,4W$. El valor obtenido es bastante elevado, pero afortunadamente, es el peor de los casos, ya que hemos considerado la corriente máxima soportada por el regulador (por tanto, la máxima que podrá consumir el circuito); en general los microcontroladores (y muchos otros dispositivos electrónicos) trabajando como "carga" consumen pulsos de corriente, por lo que la corriente promedio que suministra el regulador suele ser bastante menor, y por tanto, no se pierde tanta potencia (tanto calor) en el mismo.

Los reguladores LDO suelen presentarse físicamente en forma de encapsulados de tipo TO-220 o TO-92 (al igual que muchos transistores, curiosamente). En la práctica, esto significa que, en ambos casos, ofrecen tres patillas: una para recibir el voltaje de entrada, otra para ofrecer el voltaje de salida (que haría de "terminal positivo" para los componentes sensibles) y una tercera patilla conectada a la tierra común con la fuente de alimentación. Pero el orden y ubicación de cada patilla depende del modelo de regulador particular, así que se recomienda consultar la documentación técnica del fabricante para conocerla.

La familia de reguladores LDO más ampliamente utilizada en proyectos de electrónica doméstica es la LM78XX, donde "XX" indica el voltaje de salida (con una tolerancia cercana al 2%). El modelo concreto que más nos interesará en los proyectos donde hay una placa Arduino involucrada es el LM7805 (distribuido por Sparkfun como producto **nº 107** y por Adafruit como **nº 2164**) ya que su salida está dentro de los valores compatibles con este y puede recibir entre 7V y 35V de entrada y una intensidad máxima de 1,5A (valores comunes ofrecidos por los adaptadores AC/DC mencionados anteriormente en el apartado correspondiente). Otro modelo de regulador que también ofrece 5V ($\pm 2\%$) de salida es el L4931 (distribuido por Adafruit como producto **nº 2236**), el cual mejora el "dropout" del LM7805 porque puede recibir entre 5,5V y 20V de entrada (pero solo soporta 300mA como máximo). Si quisieramos un voltaje de salida de $3,3V \pm 1\%$ (una situación también bastante común), podríamos usar el LD1117V33 (distribuido por Sparkfun como producto **nº 526** y por Adafruit como producto **nº 2165**), el cual puede recibir entre 4,5V y 15V de entrada y una intensidad máxima de 800mA, o también podríamos echar mano del L4931-3.3 (distribuido por Adafruit como producto **nº 2166**), el cual puede recibir entre 3,7V y 20V de entrada (pero solo 250mA de intensidad máxima).

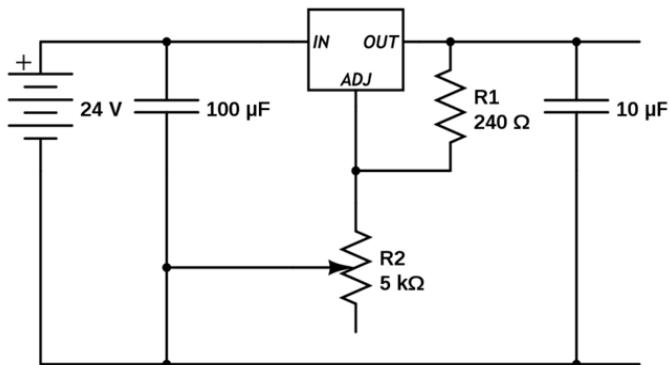
Sea cual sea el modelo de regulador, la mayoría de ocasiones veremos conectado a su patilla de entrada –y a tierra– un condensador "by-pass", y veremos conectado a su patilla de salida –y a tierra– un condensador de filtro. La razón es

eliminar las posibles oscilaciones de la señal de entrada (provocadas por ejemplo por el repentino encendido de un elemento de alto consumo de nuestro circuito, como un motor) y las de la salida (estabilizando así la tensión obtenida del regulador). Por tanto, el esquema de conexiones más común de un regulador típico (como por ejemplo el LM7805) es similar a este:



En la figura anterior se puede observar que hemos utilizado un condensador de filtro de 100 microfaradios y un condensador "by-pass" de 10 microfaradios. Estos valores suelen venir bien en la mayoría de circunstancias, así que son bastante estándares. Se podrían conectar otros condensadores en paralelo a ambos lados de diferentes capacidades, para responder a mayores variaciones del voltaje de entrada y salida respectivamente, pero por lo general, no nos encontraremos en situaciones donde esto sea necesario.

Otro modelo común de regulador es el LM317 (distribuido por Sparkfun como producto [nº 527](#)), cuya característica más interesante –además de ser capaz de soportar voltajes de entrada de entre 3V y 40V– es que permite ajustar el voltaje de salida al que nosotros deseemos (concretamente entre 1,25V y 37V, y siempre que este sea al menos 1,5V menor que el voltaje de entrada) con una intensidad máxima de salida soportada de 1,5A. Para conseguir variar ese voltaje de salida, debemos conectar al regulador un circuito auxiliar formado, además de por los dos condensadores ya vistos, por una resistencia fija (R_1) y un potenciómetro (R_2), de la siguiente manera:



EL MUNDO GENUINO-ARDUINO

Usando el LM317 y el circuito mostrado en el esquema anterior, obtendremos un voltaje de salida dado por la expresión $V_{\text{salida}}=1,25 \cdot (1 + R2/R1)$. Opcionalmente, al diseño anterior se le puede añadir un diodo para proteger el regulador contra posibles cortocircuitos en su entrada; para ello, deberíamos conectar el ánodo del diodo a la patilla de salida y el cátodo a la patilla de entrada.

Otro tipo de reguladores (construidos internamente de forma diferente a los LDO, los cuales se enmarcan dentro de la categoría de reguladores "lineales") son los llamados reguladores "switching". Estos componentes tienen como característica principal ser mucho más óptimos que los LDO en su rendimiento (es decir, en la relación "potencia transferida al circuito"/"potencia recibida de la fuente"), llegando a niveles de eficiencia del 95% (y, por tanto, disipando mucho menos calor que los LDO). No obstante, son más caros. Adafruit distribuye uno de estos reguladores "switching" (ellos los llaman "DC/DC Step-Down Converter") como producto **nº 1065** (siendo sus principales características técnicas las siguientes: 6,5-32V como rango de voltaje de entrada, $5\pm2\%$ V como voltaje de salida y 1A como intensidad máxima aportada) y otro como producto **nº 1066** (con 4,75-32V como rango de voltaje de entrada, $3,3\pm2\%$ como voltaje de salida y 1A como intensidad máxima aportada). Ambos productos tienen las mismas tres patillas (y ubicadas en el mismo orden) que el LM7805, y, además, también admiten (aunque es opcional) la misma conexión de condensadores "by-pass" y de filtro. Otro regulador "switching" (similar al primero de Adafruit pero basado en el chip LM2575) es el **SWITCHING-3TERM-5V** de Gravitech. Otro regulador interesante, esta vez comercializado en forma de "plaquita breakout", es el producto **nº 2745**: está basado en el chip LM3671, el cual tiene un rango de entrada de 3,5V a 5V (ideal para pilas LiPo), un voltaje de salida de 3,3V, una intensidad máxima aportada de 600mA y, además, un pin marcado como "Enable" que permite desconectar la salida si recibe una señal LOW.

Si quisieramos hacer uso de reguladores "switching" con voltaje de salida variable (ya sea mediante un potenciómetro incorporado o mediante la conexión a la salida de una resistencia externa), el modelo **DE-SWADJ** de la compañía Dimension Engineering, el basado en el chip **LM2596** de Tronixlabs o el producto **nº 9370** de Sparkfun (comercializado en forma de plaquita) son buenas opciones, respectivamente.

Breve nota sobre los elevadores DC/DC

En ocasiones nos puede interesar utilizar en nuestros circuitos lo contrario a un regulador de tensión; es decir, un componente que proporcione un voltaje DC de salida mayor que un determinado voltaje DC de entrada. A este elemento se le suele llamar "elevador DC/DC" o, en inglés, "DC/DC boost converter" (o también

"DC/DC step-up converter"). El "precio a pagar" en el uso de elevadores DC/DC es que la intensidad de corriente proporcionada en su salida será siempre menor que la intensidad de corriente recibida en su entrada. Ejemplos de elevadores DC/DC (comercializados en forma de plaquita) son los productos nº 10968 y nº 10967 de Sparkfun: el primero logra obtener 5V –un valor muy común en circuitos donde hay las placas Arduino– y hasta 200mA de salida a partir de un voltaje de entrada de un rango entre 1V y 4V y el segundo logra obtener 3,3V –otro valor muy común en los circuitos que estudiaremos– y hasta 200mA de salida a partir de un voltaje de entrada en un rango entre 1V y 3V de entrada. Adafruit también distribuye elevadores DC/DC en forma de plaquita; por ejemplo, su producto nº 1903 es capaz de generar 5V y 500mA a partir de un voltaje de entrada de 1,8V (o más) y su producto nº 2030 puede generar (con los mismos valores de voltaje de entrada y salida que el anterior) una intensidad de 1A.

Si queremos controlar –mediante un potenciómetro incorporado– el valor concreto (dentro de un rango) que queremos que tenga la tensión de salida ofrecida por un elevador DC/DC, podemos usar, por ejemplo, el producto de DFRobot con código **DFR0123**, el cual ofrece una tensión regulable de salida de hasta 34V (con una potencia máxima de 15W).

Es muy habitual usar los elevadores DC/DC para transformar los 3,7V de la alimentación aportada por una batería LiPo/Li-ion típica a los 5V que necesitan la mayoría de placas Arduino. Freetronics, por ejemplo, distribuye una plaquita especialmente diseñada para ello (su producto "**USB Boost 5V USB Power Supply**"), ya que por un lado consta de un zócalo JST-PH (para conectar allí este tipo de baterías) y por el otro un zócalo USB de tipo A que aporta como máximo 500mA, ofreciendo así la alimentación perfecta –a través de un cable USB A macho-<->B macho– a una placa Arduino. Sparkfun también distribuye plaquitas similares, como su producto nº 10255 o su producto nº 11231, los cuales combinan también la funcionalidad de cargador. Otros productos que también combinan a la vez la funcionalidad de elevador DC/DC con la de cargador de baterías LiPo son el nº 1944 (de 3,7V->5V a 500mA) y el nº 2465 (3,7V->5V a 1000mA), ambos de Adafruit.

Merece la pena mencionar también el producto nº 8249, portapilas de 2 pilas tipo AA y elevador DC/DC (con una salida de 5V y 200mA) todo en uno. Otro producto similar es el **DFR0250** de DFRobot, el cual ofrece igualmente una tensión de salida de 5V pero una intensidad máxima de 1A.

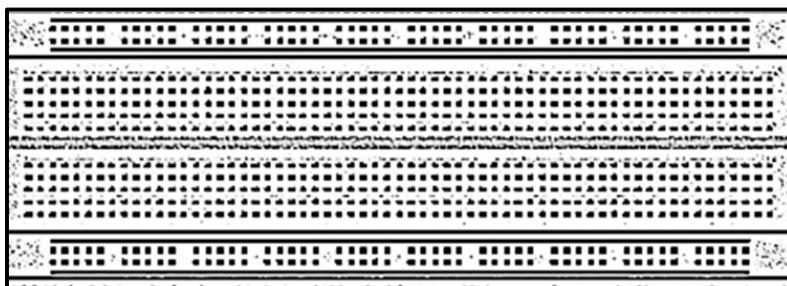
Destaquemos finalmente la posibilidad de adquirir en forma de una sola plaquita "todo en uno" tanto un elevador DC/DC como un regulador de tensión, la cual actuará de una manera u otra automáticamente dependiendo de la tensión de

entrada; estamos hablando del producto **nº 2190** de Adafruit, el cual proporciona siempre 5V bajo una tensión de entrada entre 3V y 12V.

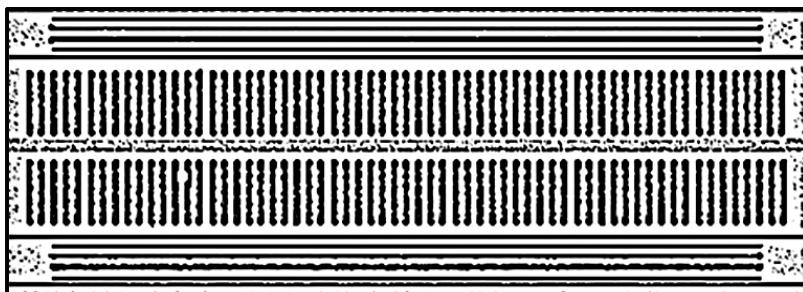
Placas de prototipado

Por "prototipado" se entiende el proceso de probar una idea de circuito creando un modelo preliminar (el "prototipo") a partir del cual se podrán derivar otros diseños más definitivos. Para construir estos prototipos es necesario que sus diferentes elementos electrónicos estén colocados sobre algún soporte común que permita de alguna manera la realización de interconexiones eléctricas entre ellos: la placa de prototipado. Existen varios tipos, pero en este apartado estudiaremos solamente las llamadas "breadboards" (también conocidas como "protoboards"), las "perfboards" y las "stripboards".

Una breadboard es una placa perforada (cuya superficie exterior es de un material aislante, generalmente plástico) en la cual cada perforación está interconectada eléctricamente en su interior con otras perforaciones concretas. De esta manera, al insertar (tantas veces como queramos) cada patilla de los diferentes componentes de nuestros circuitos en sendas perforaciones, internamente se realizarán determinadas conexiones eléctricas sin tener la necesidad de soldar nada. El objetivo es, tal como se ha dicho, poder montar prototipos rápidos pero completamente funcionales de nuestros diseños y poderlos modificar fácilmente cuando lo necesitemos. La siguiente imagen muestra la apariencia externa de una breadboard típica (que no deja de ser un conjunto de filas con agujeros):



No obstante, para poder conectar correctamente nuestros componentes a la breadboard, hemos de conocer primero cómo se estructuran sus propias conexiones internas. En este sentido, si observáramos su interior oculto bajo la superficie perforada, podríamos comprobar que está compuesto de muchas tiras de metal (normalmente cobre) dispuestas de la siguiente manera:



En la figura anterior se pueden distinguir básicamente tres zonas:

Buses: los buses se localizan en uno o ambos lados del protoboard –en la figura anterior están representados por las líneas horizontales–. Allí se conectarán los bornes de la fuente de alimentación externa. Normalmente una línea marcada con el signo "+" (y pintada la mayoría de veces de color rojo) indica el bus que será sometido al voltaje de entrada (es decir, donde insertaremos el borne positivo de la fuente) y una línea marcada con el signo "-" (y pintada la mayoría de veces de color negro o azul) representa el bus que será conectado a tierra (es decir, donde normalmente insertaremos el borne negativo). Todos los puntos del bus marcado con la línea "+" son equivalentes porque están conectados entre sí y todos los puntos del bus marcado con la línea azul también lo son entre sí, pero ambos buses están aislados eléctricamente uno del otro.

Nodos: en la parte central del protoboard aparecen gran cantidad de agujeros espaciados entre sí 0,1 pulgadas (esto es: 2,54mm). Su cantidad puede ser mayor o menor dependiendo del modelo de protoboard. Estos agujeros se usan para colocar los componentes y realizar las conexiones entre ellos. Tal como se puede observar en la figura anterior, las conexiones internas entre los agujeros están dispuestas en vertical. Lo más importante es comprender que cualquier agujero es completamente equivalente a otro que pertenezca a la misma conexión interna. Esto significa que al insertar una patilla de algún componente en un agujero, disponemos del resto de agujeros de su misma conexión interna para poder insertar en ellos una patilla de cualquier otro componente que queramos poner en contacto entre sí, tal como si los uniéramos directamente por un cable. A todos esos agujeros equivalentes conectados entre sí se les da el nombre en conjunto de "nodo". Para conectar dos nodos diferentes, lo más habitual es enchufar los extremos de un cable en un agujero de cada nodo a unir.

EL MUNDO GENUINO-ARDUINO

Canal central: es la región localizada en el medio del protoboard, que separa la zona superior de la inferior. Su presencia provoca que en cada línea vertical de agujeros haya dos nodos diferentes (es decir, sin conexión eléctrica entre sí): el de la zona superior y el de la inferior. Este canal se suele utilizar para colocar los circuitos integrados (esos componentes con forma de "cucarachas negras con patitas" también llamados "chips" o IC –del inglés "integrated circuits"–) de manera que pongamos la mitad de patitas en un lado del canal y la otra mitad en el otro lado. De esta manera, una mitad del chip estará aislada eléctricamente de la otra (tal como debe ser).

Sparkfun distribuye una protoboard de 10x30 nodos como producto **nº 12002**, Adafruit como producto **nº 64** y DFRobot como producto **FIT0096**. Todos ellos distribuyen además otra placa de 10x64 nodos como producto **nº 112**, **nº 239** y **FIT0009**, respectivamente. También es frecuente el uso de "minibreadboards", especialmente pensadas para proyectos más compactos, ya que carecen de los buses de alimentación y tierra y sus dimensiones son más reducidas (aloja tan solo 10x17 nodos); un ejemplo de ellas es el producto **nº 12043** de Sparkfun, el **nº 65** de Adafruit o el **FIT0008** de DFRobot.

Además del conocimiento de la disposición eléctrica interna de una breadboard, es importante tener en cuenta una serie de consejos útiles para el día a día que nos vendrán bien a la hora de montar nuestros diseños. Para empezar, es recomendable utilizar siempre cable negro para las conexiones a tierra, cable rojo para alimentaciones de 5V o más y cable verde (o cualquier otro color) para alimentaciones de 3V (y así evitar dañar algún componente que no admita los 5 voltios). Estos colores son simplemente una convención humana (es decir, todos los cables, tenga el color que tenga su recubrimiento de plástico, son iguales eléctricamente) pero es muy común seguirla por todo el mundo para evitar confusiones.

Otro consejo es observar si nuestra breadboard tiene pintadas las líneas roja y negra/azul de sus buses de forma no continua. Si es el caso, significa que los puntos que forman cada bus no están conectados eléctricamente todos entre sí sino que hay un salto. Para empalmar todos los puntos de cada bus y así conseguir un único bus de alimentación y un único de bus de tierra (medida aconsejable para ganar claridad y comodidad en la realización de nuestros circuitos), lo que se debe hacer es unir mediante un cable un punto de cada extremo del salto para el bus de alimentación y unir mediante otro cable un punto de cada extremo del salto para el bus de tierra (lo que se llama realizar un "puente").

CAPÍTULO 1: ELECTRÓNICA BÁSICA

Por otro lado, una precaución básica que hay que tener siempre en cuenta a la hora de utilizar el bus de tierra es la de procurar que todas las conexiones a tierra del circuito estén conectadas a su vez entre sí para que todo el circuito tenga la misma referencia (lo que a veces se llama "compartir las masas"). Es decir, que solamente exista una única tierra para todos los componentes. Esto es fundamental para que nuestros circuitos funcionen correctamente.

En el caso de protoboards que dispongan de los buses de alimentación y tierra en ambos lados (ya que existen breadboards con los buses disponibles solo en uno solo) podemos unir el bus de alimentación de un lado y del otro mediante un cable y el bus de tierra de un lado y de otro mediante otro cable. Esto nos servirá para que, al conectar la fuente de alimentación solo a un par de buses, el otro par pueda ser usado de forma equivalente también como polo positivo y negativo, siendo así más fácil y ordenada la manipulación del circuito a montar.

Finalmente, recordar que es muy importante que a la hora de añadir, quitar o cambiar componentes en una breadboard esta no reciba alimentación eléctrica alguna. Si no se hace así, se corre el riesgo de recibir una descarga y/o dañar algún componente. También es importante comprobar que las partes metálicas de los cables (u otros componentes) no contacten entre sí porque esto provocaría un cortocircuito.

Por otro lado, además de las breadboards (protoboards), existen otros tipos diferentes de placas de prototipado, de los cuales las "perfboards" y las "stripboards" son los más importantes:

Perfboards: cumplen la misma función que las breadboards, pero consiguen que el prototipo del circuito sea más sólido. Constan básicamente de una placa rígida y delgada llena de perforaciones ubicadas en forma de cuadrícula y distanciadas entre sí una distancia estándar. En estos agujeros debemos soldar nosotros los componentes de nuestro circuito y las uniones entre éstos son realizadas con cables que hemos de soldar también a la placa. Más en concreto, los componentes se colocan encima de la cara superior de la placa (por lo que sus patillas atraviesan sus agujeros, siendo soldadas estas a la cara inferior) y los cables se sueldan por la cara inferior (permaneciendo por tanto relativamente ocultos en el montaje final del circuito). Se puede identificar fácilmente la cara inferior de una "perfboard" observando la presencia de anillos metálicos rodeando a los agujeros, aunque hay que tener en cuenta que existen modelos de "perfboards" que ofrecen este tipo de agujeros anillados en ambas caras: en ese caso, cualquiera de las dos puede actuar

EL MUNDO GENUINO-ARDUINO

como cara inferior indistintamente. Ejemplos de placa con una sola cara de anillos metálicos son el producto **FIT0099** de DFRobot o el producto nº **2670** de Adafruit (el cual permite además ser troceado en partes más pequeñas según nuestras necesidades); ejemplos de placa con las dos caras de anillos metálicos son el producto **FIT0203** de DFRobot o los productos nº **8619** y nº **13268** de Sparkfun.

Stripboards: también conocidas con el nombre comercial registrado de "Veroboard", son muy similares a las perfboards. La mayor diferencia está en que mientras en una perfboard todos los agujeros estaban aislados eléctricamente entre sí (y por ello siempre se tenían que realizar las conexiones "manualmente"), en una stripboard los agujeros de una cara están unidos por líneas de cobre conductor, generalmente en forma de filas paralelas simulando la disposición de los nodos de una breadboard. De esta manera, al igual que las breadboards, las stripboards ya vienen con una serie de nodos –conjunto de agujeros conectados entre sí– predefinidos. Las únicas soldaduras que hay que realizar por tanto son las de los propios componentes y las de los cables que conecten diferentes nodos. Ejemplos de stripboards (de diferentes tamaños) son los productos nº **589**, nº **590** y nº **1609** de Adafruit o los productos nº **12702**, nº **12699** y nº **12070** de Sparkfun. Por otro lado, Sparkfun también distribuye una stripboard como producto nº **8812** que es un tanto peculiar porque todos sus agujeros están conectados entre sí formando un único nodo; esto hace que para crear nodos separados sea necesario "romper" las interconexiones metálicas pertinentes (con una herramienta cortadora tal como su producto nº **9200**) pero, a cambio, no haga falta usar cables.

Debido a que se requieren conocimientos de soldar (aunque sean mínimos) en los proyectos de este libro no usaremos ni perfboards ni stripboards. No obstante, si el lector quiere aprender a utilizar las segundas (mucho más cómodas), puede consultar un buen tutorial en la dirección <http://electronicsclub.info/stripboard.htm>.

Cables

En todos los proyectos de electrónica que trataremos en este libro (o similares), para establecer las conexiones eléctricas entre los diferentes componentes del circuito (ubicados generalmente sobre una breadboard) podremos elegir entre emplear cables sólidos ("solid") o cables trenzados ("stranded"). Los primeros están compuestos de un solo cilindro de metal y son difíciles de doblar sin que se rompan; los segundos están compuestos por múltiples hebras metálicas liadas sobre sí mismas

CAPÍTULO 1: ELECTRÓNICA BÁSICA

formando una trenza, por lo que son mucho más flexibles (siendo ideales, por tanto, para conectar partes móviles de un circuito, tales como un brazo robótico, por ejemplo). No obstante, los cables sólidos son mucho más cómodos de conectar manualmente a placas de prototipado al no deshacerse su punta en presionar sobre el nodo; por eso, elegiremos más a menudo este tipo de cables.

Otra característica que debemos conocer de un cable es la cantidad máxima de corriente que es capaz de transportar antes de dejar de funcionar correctamente o, incluso, fundirse. Esta cantidad depende de varios factores: de la composición química del cable, de su longitud, de las condiciones ambientales... pero, en general, el factor que más tendremos en cuenta aquí será su grosor: cables más gruesos pueden transportar más corriente. De hecho, el término "gauge" es usado para definir el grosor de un cable (concretamente, cada "gauge" se corresponde con un determinado diámetro y una determinada área de sección transversal). En este sentido, la industria electrónica fabrica cables con determinados "gauges" estandarizados (cuyas características eléctricas –básicamente, la cantidad máxima de corriente permitida– han sido bien estudiadas) de manera que sea fácil conocer qué tipo de cable nos interesa en cada momento. Hay dos sistemas principales para medir el gauge de un cable, el American Wire Gauge (AWG) y el Standard Wire Gauge (SWG), pero para nuestros proyectos las diferencias concretas entre ambos no serán relevantes: simplemente basta saber que los dos sistemas aumentan el número de "gauge" a medida que disminuye el diámetro (y el área de sección transversal) del cable, y, por tanto, a medida que disminuye la corriente máxima posible a transportar. En otras palabras, a mayor "gauge" del cable, mayor resistencia ofrece al paso de la electricidad. El "gauge" típico de los cables utilizados para implementar circuitos sobre breadboards es 22, ya que ese valor es el perfecto por dimensiones y características eléctricas, pero cables con otros "gauges" cercanos también pueden ser utilizados sin problemas (de hecho, para alimentar motores u otros dispositivos de alto consumo eléctrico es preferible utilizar un "gauge" de 18 o inferior).

Listar aquí la extensísima variedad de cables disponibles en Adafruit o Sparkfun (por no mencionar otros distribuidores online) no tiene mucho sentido porque su diversidad es abrumadora (más/menos largos, con más/menos "gauge", con/sin terminaciones especialmente diseñadas para ser conectados a breadboards, con diferentes tipos de recubrimiento aislante, macho-macho, macho-hembra o hembra-hembra, etc.). Si el lector quiere conocer, de todas formas, qué es lo que ofrecen estas empresas, se puede consultar la lista completa accediendo a la categoría "Wires" de sus respectivas tiendas online; concretamente, aquí (<https://www.sparkfun.com/categories/141>) o aquí (<http://www.adafruit.com/category/125>).

EL MUNDO GENUINO-ARDUINO

Por otro lado, mencionar que si en algún momento necesitáramos quitar de un extremo de un cable su capa protectora aislante de plástico que lo recubre (para, por ejemplo, poder conectar eléctricamente su cilindro metálico interno con otro componente), será de gran ayuda un "pelador" de cables, tal como el producto nº 527 de Adafruit o el nº 12630 de Sparkfun, ambos compatibles con los "gauges" más populares.

Si lo que necesitáramos fuera unir dos cables pero sin tener que soldarlos, podemos emplear los productos nº 1496, nº 866 o nº 874 de Adafruit, los cuales son simples conectores que actúan como nodo común para hasta dos, tres o cinco cables, respectivamente. Eso sí, es recomendable usar unos alicates ajustables para insertar los cables convenientemente dentro del conector.

USO DE UNA PLACA DE PROTOTIPADO

En este apartado aparecen recopilados algunos ejemplos que muestran diferentes maneras de conectar dispositivos mediante una "breadboard".

Breve nota sobre cómo alimentar circuitos en placas de prototipado

En los siguientes diagramas se muestra una manera concreta de alimentar los circuitos de ejemplo (mediante pilas AA incluidas en un portapilas con los cables-borne al aire) que no tiene por qué ser, ni mucho menos, la única manera de conectar una fuente a una breadboard. Por ejemplo, si dispusiéramos de un adaptador AC/DC como los mencionados en un apartado anterior, podríamos conseguir una alimentación a unos niveles de tensión compatibles con la mayoría de elementos electrónicos (es decir, 3,3V o 5V) si nos ayudamos de un regulador de tensión y un par de condensadores y resistencias complementarios (tal como muestran, de hecho, los esquemas eléctricos indicados en el apartado sobre reguladores); precisamente el producto nº 8373 de Sparkfun aporta todos estos componentes necesarios (regulador, condensadores, etc.) para implementar este circuito alimentador-regulador.

Uno de los componentes incluidos en el producto anterior es el zócalo 5,5/2,1mm donde se debe enchufar el adaptador AC/DC. Este componente (que se distribuye por separado para nuestra conveniencia con código nº 10811 en Sparkfun y nº 373 en Adafruit) dispone de una patilla metálica posterior que debe funcionar como borne positivo (así que, por tanto, el cable de la alimentación deberá ser conectado a ella) y otra patilla metálica frontal que debe funcionar como borne negativo (así que, por tanto, el cable de tierra deberá ser conectado a ella); también dispone de

una patilla lateral pero no nos será relevante. De todas formas, una alternativa al zócalo anterior es el zócalo **nº 368** de Adafruit, que permite conectar ambos bornes de una forma más sencilla.

No obstante, si queremos una solución algo más cómoda, existen pequeñas plaquitas ya prediseñadas con todos los componentes necesarios (incluyendo el zócalo 5,5/2,1mm) listas para ser conectadas directamente a una breadboard y no tener así que implementar el circuito alimentador-regulador "a mano"; ejemplos de estas plaquitas son los productos **nº 13032** o **nº 114** de Sparkfun o el **nº 184** de Adafruit (los dos últimos distribuidos en forma de kit para ser ensamblados en casa). También son destacables, en este sentido, los **Power supplies** de Akafugu, el **Breadboard Power Supply Module** de IteadStudio o el **Breadboard Power Supply Kit** de Gravitech: todos ellos permiten regular a 5V o a 3,3V según sea necesario.

Por otro lado, si disponemos de baterías LiPo/Li-ion puede ser interesante el producto **nº 1863** de Adafruit (una plaquita con un zócalo JST-PH que está preparada por el otro extremo para ser conectada fácilmente a una breadboard). Si, en cambio, disponemos de pilas CR2032, los productos de Adafruit **nº 1870** –sin interruptor incorporado– o **nº 1871** –con interruptor incorporado– pueden ser lo más práctico al ser plaquitas que ofrecen un soporte para este tipo de pilas a la vez que también permiten ser conectadas fácilmente a una breadboard por el otro extremo.

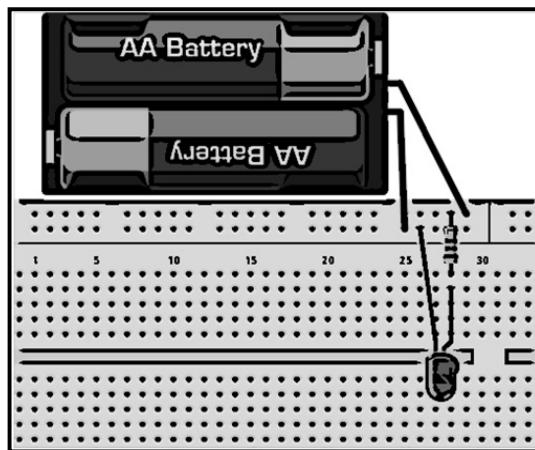
Si aún seguimos queriendo alimentar directamente nuestra breadboard mediante cables-borne al aire pertenecientes a una fuente externa (como muestran los diagramas siguientes), puede sernos útil el producto **nº 1400** de Adafruit, consistente en una plaquita que por un lado recibe los dos cables de la fuente y por otro los ofrece al resto del circuito, y cuya función es permitir cortar la corriente (de 3V a 14V y 3A como máximo) y volverla a dejar fluir con tan solo pulsar el botón-interruptor que lleva incorporado.

En el caso de que sean perfboards o stripboards lo que queramos alimentar mediante los cables-borne de una fuente externa no resulta buena idea soldar dichos cables a la placa en cuestión porque se impide el reemplazo cómodo de la fuente asociada. En esos casos lo que se suele soldar a la placa son unos componentes llamados "terminales de tornillo" (en inglés, "screw terminals"), los cuales ofrecen puntos de enchufe donde se pueden conectar los cables necesarios (asegurándolos mediante presión) y desconectar muy fácilmente. Ejemplos de ellos son los productos **nº 1074** o **nº 1081** de Adafruit o el **nº 10571** de Sparkfun (todos ellos compatibles con un gauge de cable de 22AWG –el más habitual en nuestros

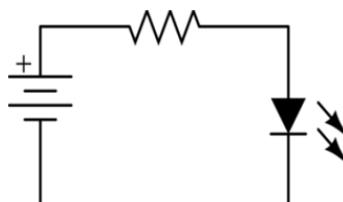
EL MUNDO GENUINO-ARDUINO

proyectos— y con el espaciado entre los orificios de las perfboards/stripboards estándar —al ofrecer una separación entre pines de 0,1"—).

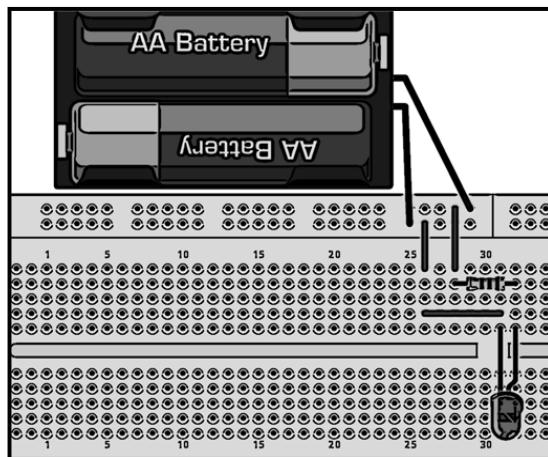
Ejemplo nº 1: en el siguiente diagrama se puede ver cómo se realiza un circuito donde una resistencia y un LED se conectan en serie.



En el dibujo anterior el polo positivo de la fuente de alimentación se conecta al bus más exterior. Esto hará que cualquier elemento conectado a algún agujero de ese bus reciba directamente de allí la alimentación eléctrica. Eso es lo que le pasa precisamente a la resistencia de nuestro circuito: su terminal superior está enchufado en ese bus. El otro terminal está colocado en un nodo de la breadboard, nodo donde precisamente se conecta también el terminal positivo del LED (en el diagrama, es el que aparece con una pequeña hendidura en su raíz). Esto quiere decir que estos dos componentes están directamente conectados. Finalmente, el terminal negativo del LED está enchufado en el bus más interior de la breadboard, bus en el cual se conecta también el polo negativo de la fuente, por lo que el LED está directamente conectado a él, cerrando el círculo. El circuito anterior tiene un esquema como este:

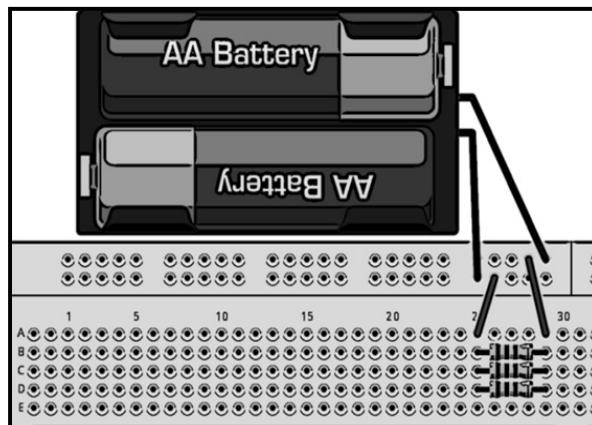


También se podría haber hecho el mismo circuito llevando el cable de alimentación y el de tierra a la zona de nodos:

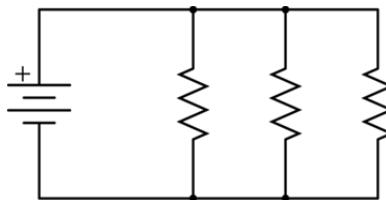


Es importante tener en cuenta que todos los agujeros de un mismo nodo representan un único punto de conexión (un error muy común al principio es conectar ambos terminales de un dispositivo en un mismo nodo, cosa que no tiene sentido). Sabiendo esto, es fácil ver en la ilustración anterior que el terminal izquierdo de la resistencia está conectado a un cable que a su vez está conectado a la alimentación, y que el terminal derecho de la resistencia está conectado al terminal positivo del LED mientras que su terminal negativo está conectado a un cable que a su vez está conectado a otro, el cual va a parar finalmente a tierra.

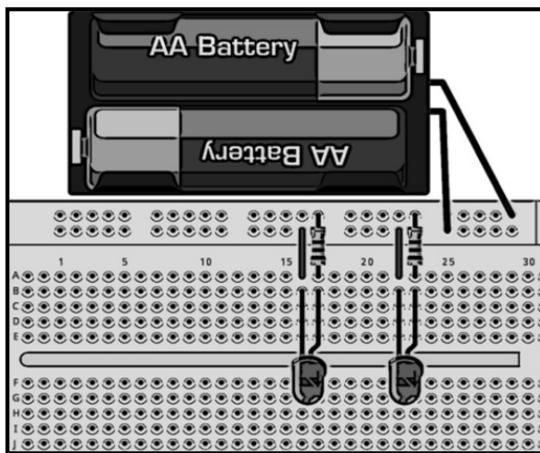
Ejemplo nº 2: la siguiente ilustración muestra la conexión de tres dispositivos en paralelo (concretamente, tres resistencias):



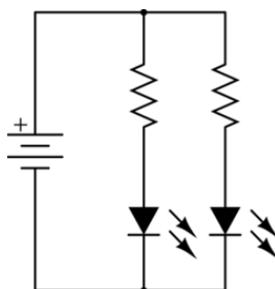
Su esquema equivalente sería:



Ejemplo nº 3: el siguiente esquema muestra la conexión en serie de una resistencia y un LED, y la conexión en paralelo de ambos a otra resistencia y LED.

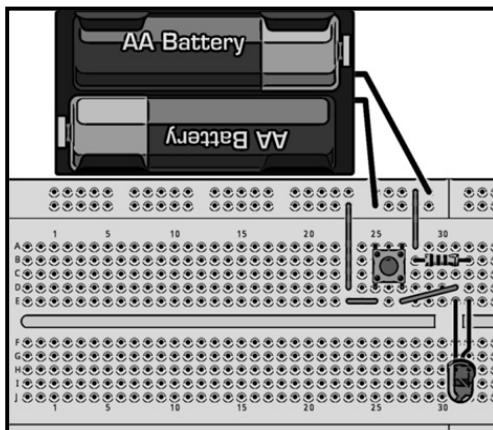


Si probamos el circuito anterior en la realidad, veremos que dependiendo del valor de cada resistencia, el LED correspondiente se iluminará más o menos. Además, también podremos observar que, en cualquier caso, ambos LEDs se iluminarán siempre a menos que si estuvieran conectados en serie (o si solo hubiera un LED en vez de dos): esto es porque, como ya sabemos, en las conexiones en paralelo el flujo de corriente se ha de "dividir" por todos los caminos posibles, repartiéndose por tanto la intensidad entre ellos. En cualquier caso, su esquema eléctrico es el siguiente:

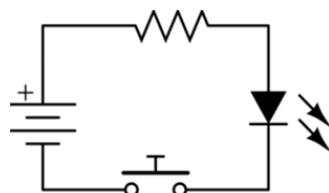


CAPÍTULO 1: ELECTRÓNICA BÁSICA

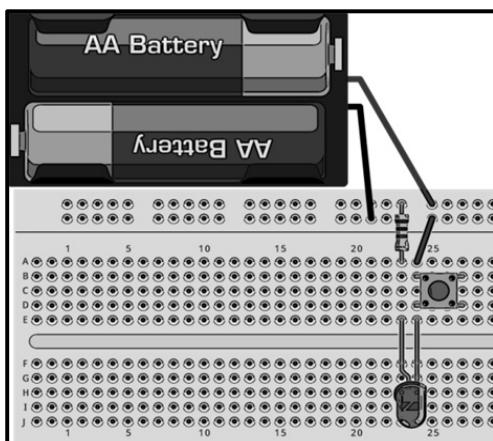
Ejemplo nº 4: el siguiente dibujo muestra la conexión de tres dispositivos en serie: un LED, una resistencia y un pulsador:



El esquema equivalente del circuito anterior sería:

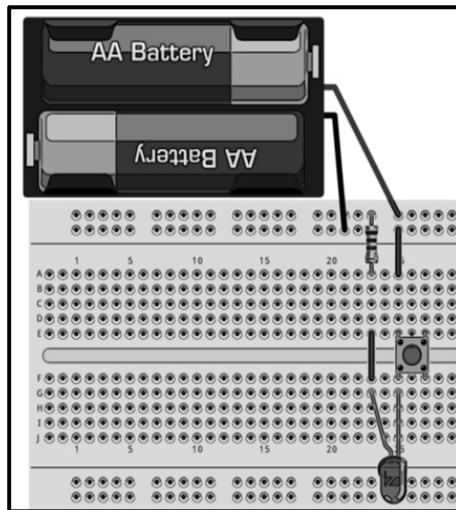


El mismo circuito podría haber sido implementado, de hecho, sin tanto cable:

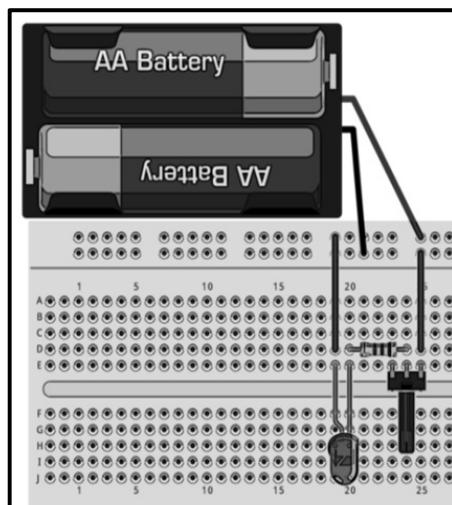


EL MUNDO GENUINO-ARDUINO

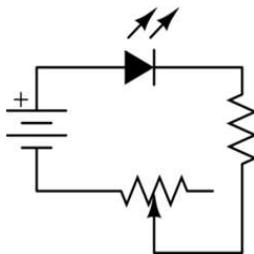
Es importante observar las figuras anteriores y fijarse en cómo están ubicadas las patillas del pulsador. Si en el último circuito hubiéramos colocado el pulsador de la manera que se muestra en la siguiente figura, el LED estaría encendido siempre porque, independientemente del estado del pulsador, las dos patillas enfrentadas siempre están unidas internamente (ya que en realidad, representan un único punto de conexión, tal como ya hemos estudiado).



Ejemplo nº 5: el siguiente esquema muestra la conexión de tres dispositivos en serie: un LED, una resistencia y un potenciómetro:



Su esquema correspondiente es este (donde podemos ver claramente que se conecta la patilla central del potenciómetro y uno de sus extremos, pero no el otro):



Tenemos que imaginar que la flecha central del símbolo del potenciómetro se moverá desde un extremo hasta el otro de ese símbolo según giremos la rosca del potenciómetro. Tal como está dibujado el esquema anterior, si la flecha se "sitúa" en el extremo derecho del símbolo, el potenciómetro funcionará con su valor de resistencia máximo; si la flecha se "sitúa" en el extremo izquierdo, el potenciómetro no ofrecerá resistencia alguna.

Este último hecho es la causa de haber añadido una resistencia entre el LED y el potenciómetro: si en algún momento ajustáramos el potenciómetro a cero, ¡no habría ninguna resistencia en el circuito!, y esto dañaría irreversiblemente el LED porque recibiría demasiada intensidad de corriente. Por tanto, la resistencia adicional mantiene un valor mínimo que no es rebajado nunca.

USO DE UN MULTÍMETRO DIGITAL



El multímetro digital es un instrumento que sirve básicamente para medir alguna de las tres magnitudes relacionadas por la Ley de Ohm: o bien el voltaje existente entre dos puntos de un circuito, o bien la intensidad de corriente que fluye a través de él, o bien la resistencia que ofrece cierto componente. Dependiendo del modelo, también hay multímetros que pueden medir otras magnitudes como la capacidad de condensadores, la ganancia de transistores BJT y más. Es decir, es una herramienta que nos permite comprobar el correcto funcionamiento de los componentes y circuitos electrónicos, por lo que es fundamental tenerla a mano cuando realicemos nuestros proyectos.

Existen muchos modelos diferentes de multímetros digitales, por lo que es importante leer el manual de instrucciones del

EL MUNDO GENUINO-ARDUINO

fabricante para asegurar el buen funcionamiento del instrumento. Ejemplos de multímetros son el **producto nº 12966** de Sparkfun o los productos **nº 2610, nº 2034** y **nº 308** de Adafruit. Aunque dependiendo del modelo puedan cambiar la posición de sus elementos y la cantidad de funciones, en general podemos identificar las partes y funciones estándar de un multímetro genérico como las siguientes:

Botón de "power" (apagado-encendido): la mayoría de multímetros son alimentados mediante pilas.

Display: pantalla de cristal líquido en donde se mostrarán los resultados de las mediciones.

Llave selectora: sirve para elegir el tipo de magnitud a medir y el rango de medición. Los símbolos que la rodean indican el tipo de magnitud a medir, y los más comunes son el voltaje directo ($V-$) y alterno (V^{\sim}), la corriente directa ($A-$) y alterna (A^{\sim}), la resistencia (Ω) o la capacidad (F). Los números que rodean la llave indican el rango de medición: el rango que hayamos seleccionado nos indica el valor máximo (de la magnitud correspondiente) que podemos medir de forma segura. Por tanto, cuando no sepamos a priori el valor de la magnitud a medir, deberemos situar el multímetro en el rango máximo e ir bajando para efectuar medidas cada vez más precisas, pero con cuidado de no situarlo en un rango inferior al adecuado para esa medida (en el display se irán mostrando los valores numéricos medidos de acuerdo a la escala elegida).

Cables (habitualmente) rojo y negro con punta: el cable negro siempre se conectará al zócalo negro del multímetro (solo existe uno, y generalmente está señalado con la palabra "COM" –de "referencia COMún"–), mientras que el cable rojo se conectará al zócalo rojo adecuado según la magnitud que se quiera medir (ya que hay varios): si se quiere medir voltaje o resistencia (tanto en continua como en alterna), se deberá conectar el cable rojo al zócalo rojo marcado normalmente con el símbolo " $+V\Omega$ " ; si se quiere medir intensidad de corriente (tanto en continua como en alterna), se deberá conectar el cable rojo al zócalo rojo marcado con el símbolo "mA" o bien "A", dependiendo del rango a medir.

Una vez conocidas las partes funcionales de esta herramienta, la podemos utilizar para realizar diferentes medidas:

Para medir el voltaje (continuo) existente entre dos puntos de un circuito alimentado, deberemos conectar los cables convenientemente al multímetro para colocar a continuación la punta del cable negro en un punto del circuito y la del cable rojo en el otro (de tal forma que en realidad estemos realizando una conexión en paralelo con dicho circuito). Seguidamente, moveremos la llave selectora al símbolo V- y elegiremos el rango de medición adecuado. Si este lo desconocemos, lo que debemos hacer, tal como ya hemos dicho, es empezar por el rango más elevado e ir bajando paso a paso para obtener finalmente la precisión deseada. Si bajamos más de la cuenta (es decir, si el valor a medir es mayor que el rango elegido), lo sabremos porque a la izquierda del display se mostrará el valor especial "1".

También podríamos utilizar la posibilidad que ofrece el multímetro de medir voltaje continuo para conocer la diferencia de potencial generada por una determinada fuente de alimentación (y así saber en el caso de una pila, por ejemplo, si está gastada o no). En este caso, deberíamos colocar la punta del cable rojo en el borne positivo de la fuente y el negro en el negativo y proceder de la misma manera, seleccionando la magnitud y rango a medir. Por ejemplo, si quisieramos comprobar que una pila de 1,5V nominales genera efectivamente esa tensión, deberíamos situar el rango del multímetro en la posición estandarizada de "20V". Si quisieramos comprobar que un determinado adaptador AC/DC genera realmente una tensión de 9V (por ejemplo), deberíamos situar el rango del multímetro también en "20V".

Para medir la resistencia de un componente, debemos mantener desconectado dicho componente para que no reciba corriente de ningún circuito. El procedimiento para medir una resistencia es bastante similar al de medir tensiones: basta con conectar cada terminal del componente a los cables del multímetro (si el componente tiene polaridad, como es el caso de los diodos y de algunos condensadores, el cable rojo se ha de conectar al terminal positivo del componente y el negro al negativo; si el componente no tiene polaridad, esto es indiferente) y colocar el selector en la posición de ohmios y en el rango a priori más apropiado al valor de la resistencia que se desea medir. Si no sabemos qué rango elegir, empezaremos colocando la ruleta en el rango más grande; si este resultara ser demasiado elevado para el valor concreto que tenga la resistencia a medir, el display del multímetro mostrará una cifra muy cercana a 0 porque no hay suficiente resolución; en ese caso, deberemos ir reduciendo el rango hasta que encontremos el que más precisión nos dé (por ejemplo, para medir una resistencia de 330Ω , deberemos seleccionar el rango estandarizado de "2k Ω ", obteniendo en el display la cifra 0,330 aproximadamente). Si el rango elegido llegara a ser

EL MUNDO GENUINO-ARDUINO

menor que el valor a medir, el display mostrará entonces un "1" a su izquierda; en ese caso, por tanto, habría que ir aumentándolo hasta encontrar el correcto.

Para medir la intensidad que fluye por un circuito, hay que conectar el multímetro en serie con el circuito en cuestión. Por eso, para medir intensidades tendremos que abrir el circuito para intercalar el multímetro en medio, con el propósito de que la intensidad circule por su interior. Concretamente, el proceso a seguir es: insertar el cable rojo en el zócalo adecuado (mA o A según la cantidad de corriente a medir, ¡esto es importante para no dañar el multímetro!) y el cable negro en el zócalo negro, empalmar cada cable del multímetro en cada uno de los dos extremos del circuito abierto que tengamos (cerrándolo así, por lo tanto) y ajustar el selector a la magnitud y rango adecuados.

Idealmente, un multímetro funcionando como medidor de corriente tiene una resistencia nula al paso de la corriente a través de él (precisamente para evitar alteraciones en la medida del valor de la intensidad real), por lo que está relativamente desprotegido de intensidades muy elevadas y pueda dañarse con facilidad. Hay que tener siempre en cuenta por tanto el máximo de corriente que puede soportar, el cual lo ha de indicar el fabricante (además del tiempo máximo que puede estar funcionando en este modo).

Para medir continuidad (es decir para comprobar si dos puntos de un circuito están eléctricamente conectados), simplemente se debe ajustar el selector en la posición marcada con el signo de una "onda de audio" y conectar los dos cables a cada punto a medir (no importa la polaridad). Atención: este modo solo se puede utilizar cuando el circuito a medir no está recibiendo alimentación eléctrica. Si hay continuidad, el multímetro emitirá un sonido (gracias a un zumbador que lleva incorporado); si no, no se escuchará nada. También se puede observar lo que muestra el display según el caso, pero el mensaje concreto depende del modelo, así que se recomienda consultar las instrucciones de cada aparato en particular.

Una aplicación práctica del multímetro funcionando como medidor de continuidad es la comprobación de qué agujeros de una breadboard pertenecientes al mismo nodo mantienen su conectividad, ya que después de un uso continuado es relativamente fácil que esta se estropee.

Para medir la capacidad de un condensador también podemos utilizar muchos multímetros digitales del mercado. Tan solo tendremos que conectar las patillas del condensador a unos zócalos especiales para ello, marcados con la marca "CX". Los condensadores deben estar descargados antes de conectarlos a dichos zócalos. Para los condensadores que tengan polaridad habrá que identificar el zócalo correspondiente a cada polo en el manual del fabricante.

Para conocer la polaridad de un LED, se debe ajustar el selector en la posición marcada con el símbolo de un diodo y conectar los dos cables del multímetro a cada borne del LED a estudiar: si este se ilumina, el cable positivo del multímetro está tocando el ánodo, y el negativo, el cátodo; si no, es al revés.

HARDWARE GENUINO



¿QUÉ ES UN SISTEMA ELECTRÓNICO?

Podemos definir "sistema electrónico" como un conjunto de cuatro partes: sensores, circuitería de procesamiento y control, actuadores y fuente de alimentación.

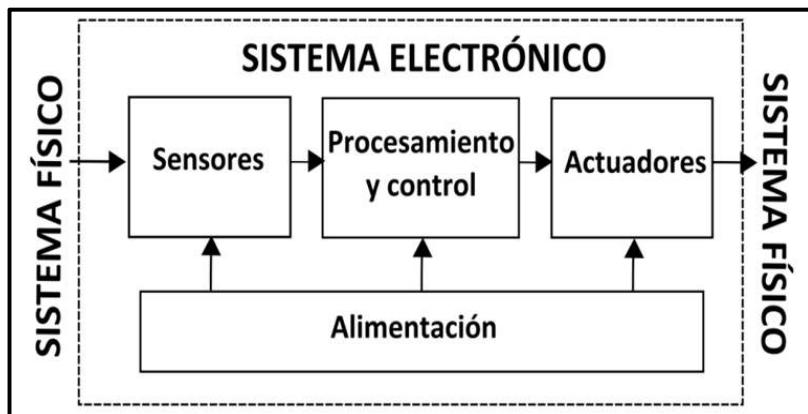
Los sensores obtienen información del mundo físico externo y la transforman en una señal eléctrica que puede ser manipulada por la circuitería interna de control. Existen sensores de todo tipo: de temperatura, de humedad, de movimiento, de luz, de sonido (micrófonos), etc.

Los circuitos internos de un sistema electrónico procesan la señal eléctrica convenientemente. La manipulación de dicha señal dependerá tanto del diseño de los diferentes componentes hardware del sistema, como del conjunto lógico de instrucciones (es decir, del "programa") que dicho hardware tenga pregrabado y que sea capaz de ejecutar de forma autónoma.

Los actuadores transforman la señal eléctrica acabada de procesar por la circuitería interna en energía que actúa directamente sobre el mundo físico externo. Ejemplos de actuadores son: un motor (energía mecánica), una bombilla (energía lumínica), un altavoz (energía acústica), etc.

EL MUNDO GENUINO-ARDUINO

La fuente de alimentación proporciona la energía necesaria para que se pueda realizar todo el proceso recién descrito de "obtención de información del medio <-> procesamiento <-> actuación sobre el medio". Ejemplos de fuentes son las pilas, baterías, adaptadores AC/DC, etc.



¿QUÉ ES UN MICROCONTROLADOR?

Un microcontrolador es un circuito integrado o "chip" (es decir, un dispositivo electrónico que integra dentro de un único encapsulado un gran número de componentes –resistencias, condensadores, transistores, etc. – conectados entre sí de forma muy específica) que tiene la característica de ser programable. Es decir, que es capaz de ejecutar de forma autónoma una serie de instrucciones previamente definidas por nosotros. En el diagrama anterior, representativo de un sistema electrónico, el microcontrolador sería el componente principal de la circuitería de procesamiento y control.

Por definición, un microcontrolador (también llamado comúnmente "micro") ha de incluir en su interior tres elementos básicos:

CPU (Unidad Central de Proceso): es la parte encargada de ejecutar cada instrucción y de controlar que dicha ejecución se realice correctamente. Normalmente, una instrucción hace uso de datos disponibles previamente (los "datos de entrada"), y genera como resultado otros datos diferentes (los "datos de salida"), que podrán ser utilizados (o no) por la siguiente instrucción.

Diferentes tipos de memorias: son las encargadas de alojar tanto las instrucciones como los diferentes datos que estas necesitan. De esta manera posibilitan que toda esta información (instrucciones y datos) esté siempre disponible para que la CPU pueda acceder y trabajar con ella en cualquier momento. Generalmente encontraremos dos tipos de memorias: las que su contenido se almacena de forma permanente incluso tras cortes de alimentación eléctrica (llamadas "persistentes"), y las que su contenido se pierde al dejar de recibir alimentación (llamadas "volátiles"). Según las características de la información a guardar, habitualmente esta se grabará de forma automática en un tipo u otro de memoria.

Diferentes patillas de E/S (entrada/salida): son las encargadas de comunicar el microcontrolador con el exterior. En las patillas de entrada del microcontrolador podremos conectar sensores para que este pueda recibir datos provenientes de su entorno, y en sus patillas de salida podremos conectar actuadores para que el microcontrolador pueda enviarles órdenes y así interactuar con el medio físico. De todas formas, muchas patillas de la mayoría de microcontroladores no son exclusivamente de entrada o de salida, sino que pueden ser utilizados indistintamente para ambos propósitos (de ahí el nombre de E/S).

Es decir, un microcontrolador es un chip cuya CPU está especializada en ejecutar constantemente un conjunto de instrucciones que han sido previamente guardadas en una de sus memorias. Lógicamente, estas instrucciones serán diferentes según el uso que se le quiera dar al microcontrolador, y deberemos de decidir nosotros cuáles son; muchas de ellas sirven para reconocer la información proveniente de distintos sensores (y recibida a través de las patillas de E/S conectadas a ellos), otras sirven para realizar cálculos matemáticos con esa (u otra) información, otras sirven para reaccionar en consecuencia y operar sobre diferentes actuadores (a través de las patillas de E/S conectadas a ellos), etc., etc.

Cada vez existen más productos domésticos que incorporan algún tipo de microcontrolador con el fin de aumentar sustancialmente sus prestaciones, reducir su tamaño y coste, mejorar su fiabilidad y disminuir el consumo. Así, podemos encontrar microcontroladores dentro de multitud de dispositivos electrónicos que usamos en nuestra vida diaria, como pueden ser desde un simple timbre hasta un completo robot pasando por juguetes, frigoríficos, televisores, lavadoras, microondas, impresoras, el sistema de arranque de nuestro coche, etc.

¿QUÉ ES GENUINO/ARDUINO?

Arduino (cuya web oficial es <http://www.arduino.cc> o, también, <http://www.genuino.cc>) es, en realidad, tres cosas:

Varios modelos de placas de hardware libre que incorporan, cada uno de ellos, un determinado modelo de microcontrolador reprogramable y una serie de pines-hembra (unidos internamente a las patillas de E/S de dicho microcontrolador) que permiten conectar allí de forma muy sencilla y cómoda diferentes sensores y actuadores.

NOTA: Por motivos explicados en la nota aclaratoria de la introducción de este libro, a estas placas también se les da el nombre alternativo de "Genuino". Nosotros, de todas formas, las seguiremos llamando "Arduino" en este libro.

Cuando hablamos de "placa de hardware" nos estamos refiriendo en concreto a una PCB (del inglés "printed circuit board", que literalmente significa "placa de circuito impreso"). Las PCBs son superficies fabricadas de un material no conductor (normalmente resinas de fibra de vidrio reforzada, cerámica o plástico) sobre las cuales aparecen laminadas ("pegadas") pistas de material conductor (normalmente cobre). Las PCBs se utilizan para conectar eléctricamente, a través de los caminos conductores, diferentes componentes electrónicos soldados a ella. Una PCB es la forma más compacta y estable de construir un circuito electrónico (en contraposición a una breadboard, perfboard o similar) pero, al contrario que estas, una vez fabricada, su diseño es bastante difícil de modificar. Así pues, una placa Arduino no es más que una PCB que implementa un determinado diseño de circuitería interna.

El diseño hardware de la placa Arduino original estuvo inspirado en el de otra placa de hardware libre preexistente, la placa Wiring (<http://www.wiring.co>). Esta placa surgió en 2003 como proyecto personal de Hernando Barragán, estudiante por aquel entonces del Instituto de Diseño de Ivrea (lugar donde surgió en 2005 precisamente la primera placa Arduino). No obstante, hoy en día hablar genéricamente de "placa Arduino" no tiene mucho sentido porque, tal como ya hemos indicado, actualmente existen varios modelos oficiales de placas Arduino, cada una con diferentes características (como el modelo concreto de microcontrolador incorporado –y como consecuencia, entre otras cosas, la cantidad de memoria utilizable–, el tipo de pines-hembra ofrecidos, el tamaño físico, etc., etc.). Por tanto, antes de empezar cualquier proyecto conviene estudiar bien esas características (comparando las de los diferentes modelos entre sí) para poder elegir la placa Arduino más acorde a nuestras necesidades. En el siguiente apartado conoceremos más a fondo los distintos tipos de placas existentes.

Un software (más en concreto, un "entorno de desarrollo") **gratis, libre y multiplataforma** (ya que funciona en Linux, OS X y Windows) que debemos instalar en nuestro ordenador y que nos permite escribir, verificar y guardar ("cargar") en la memoria del microcontrolador de cualquier placa Arduino el conjunto de instrucciones que deseamos que este empiece a ejecutar. Es decir: nos permite programarlo. La manera estándar de conectar nuestro computador con una placa Arduino para poder enviarle y grabarle dichas instrucciones es mediante un simple cable USB, gracias a que la mayoría de placas Arduino incorporan un conector de este tipo.

Los proyectos Arduino pueden ser autónomos o no. En el primer caso, una vez programado su microcontrolador, la placa no necesita estar conectada a ningún computador y puede funcionar autónomamente si dispone de alguna fuente de alimentación. En el segundo caso, la placa debe estar conectada de alguna forma permanente (por cable USB, por cable de red Ethernet, etc.) a un computador ejecutando algún software específico que permita la comunicación entre este y la placa y el intercambio de datos entre ambos dispositivos. Este software específico lo deberemos programar generalmente nosotros mismos mediante algún lenguaje de programación estándar como Python, C, Java, Php, etc., y será independiente completamente del entorno de desarrollo Arduino, el cual no se necesitará más, una vez que la placa ya haya sido programada y esté en funcionamiento.

Un lenguaje de programación libre. Por "lenguaje de programación" se entiende cualquier idioma artificial diseñado para expresar instrucciones (siguiendo unas determinadas reglas sintácticas) que pueden ser llevadas a cabo por máquinas. Concretamente dentro del lenguaje Arduino, encontramos elementos parecidos a muchos otros lenguajes de programación existentes (como los bloques condicionales, los bloques repetitivos, las variables, etc.), así como también diferentes comandos –asimismo llamados "órdenes" o "funciones"– que nos permiten especificar de una forma coherente y sin errores las instrucciones exactas que queremos programar en el microcontrolador de cualquier placa Arduino. Estos comandos los escribimos usando el entorno de desarrollo Arduino.

Tanto el entorno de desarrollo como el propio lenguaje de programación Arduino estuvieron inspirados originalmente en otro entorno y lenguaje libre preexistente llamado Processing (<http://www.processing.org>), desarrollado inicialmente por Ben Fry y Casey Reas. Processing es un entorno y lenguaje especializado en facilitar la generación de imágenes en tiempo real, de animaciones 2D/3D y de interacciones visuales, por lo que muchos profesores del Instituto de

EL MUNDO GENUINO-ARDUINO

Diseño de Ivrea lo utilizaban en sus clases. De ahí que fuera el punto de partida más natural (por familiaridad y experiencia) para desarrollar lo que acabaría siendo el entorno y lenguaje Arduino.

Con Arduino se pueden realizar multitud de proyectos de rango muy variado: desde robótica hasta domótica, pasando por monitorización de sensores ambientales, sistemas de navegación, aplicaciones de automatización industrial, telemática, etc. Realmente, las posibilidades de esta plataforma para el desarrollo de productos electrónicos son prácticamente infinitas y tan solo están limitadas por nuestra imaginación.

¿CUÁL ES EL ORIGEN DE ARDUINO?

Arduino nació en el año 2005 en el Instituto de Diseño Interactivo de Ivrea (Italia), centro académico donde los estudiantes se dedicaban a experimentar con la interacción entre humanos y diferentes dispositivos (muchos de ellos basados en microcontroladores) para conseguir generar espacios únicos, especialmente artísticos. Arduino apareció por la necesidad de contar con un dispositivo para utilizar en las aulas que fuera de bajo coste, que funcionase bajo cualquier sistema operativo y que contase con documentación adaptada a gente que quisiera empezar de cero. La idea original fue, pues, fabricar la placa para uso interno de la escuela.

No obstante, el Instituto se vio obligado a cerrar sus puertas precisamente en 2005. Ante la perspectiva de perder en el olvido todo el desarrollo del proyecto Arduino que se había ido llevando a cabo durante aquel tiempo, se decidió liberarlo y abrirlo a "la comunidad" para que todo el mundo tuviera la posibilidad de participar en la evolución del proyecto, proponer mejoras y sugerencias y mantenerlo "vivo". Y así ha sido: la colaboración de muchísima gente ha hecho que Arduino poco a poco haya llegado a ser lo que es actualmente: un proyecto de hardware y software libre de ámbito mundial.

El principal responsable de la idea y diseño de Arduino, y la cabeza visible del proyecto es el llamado "Arduino Team", formado por Massimo Banzi (profesor en aquella época del Instituto Ivrea), David Cuartielles (profesor de la Escuela de Artes y Comunicación de la Universidad de Malmö, Suecia), David Mellis (por aquel entonces estudiante en Ivrea y actualmente miembro del grupo de investigación High-Low Tech del MIT Media Lab) y Tom Igoe (profesor de la Escuela de Arte Tisch de Nueva York). También formaba parte originalmente del "Arduino Team" Gianluca Martino, propietario de la empresa responsable de la fabricación a escala industrial de las

placas; no obstante, tal como explica la nota aclaratoria sobre "Genuino vs. Arduino" de la introducción de este libro, en 2015 se rompieron las relaciones entre la empresa de Gianluca y el resto del "Arduino Team", de forma que este último actualmente ha establecido contratos de colaboración con diferentes fabricantes de hardware del mundo (ofreciendo todos ellos las mismas placas pero bajo el nombre de "Genuino") mientras que la primera sigue autodenominándose "Arduino" (abriendo así una guerra comercial que no tiene final visible). En este libro, tal como ya se comentó en la nota citada, nos centraremos en el proyecto del "Arduino Team".

Existe un documental de 30 minutos muy interesante, en el cual interviene el "Arduino Team" (antes de la escisión) explicando en primera persona todo el proceso de gestación y evolución del proyecto Arduino, desde los detalles técnicos que se tuvieron en cuenta hasta la filosofía libre que impregnó (e impregna) su desarrollo, pasando por diferentes testimonios de colaboradores de todo el mundo. Se puede ver gratuitamente en <https://vimeo.com/18390711>.

¿QUÉ QUIERE DECIR QUE ARDUINO SEA "SOFTWARE LIBRE"?

En párrafos anteriores hemos comentado que Arduino es una placa de "hardware *libre*" y también que es "un entorno y lenguaje de programación (es decir, software) *libre*". ¿Pero qué significa aquí la palabra "libre" exactamente? Según la Free Software Foundation (<http://www.fsf.org>), organización encargada de fomentar el uso y desarrollo del software libre a nivel mundial, un software para ser considerado libre ha de ofrecer a cualquier persona u organización cuatro libertades básicas e imprescindibles:

Libertad 0: la libertad de usar el programa con cualquier propósito y en cualquier sistema informático.

Libertad 1: la libertad de estudiar cómo funciona internamente el programa, y adaptarlo a las necesidades particulares. El acceso al código fuente es un requisito previo para esto.

Libertad 2: la libertad de distribuir copias.

Libertad 3: la libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie. El acceso al código fuente es un requisito previo para esto.

EL MUNDO GENUINO-ARDUINO

Un programa es software libre si los usuarios tienen todas estas libertades. Así pues, el software libre es aquel software que da a los usuarios la libertad de poder ejecutarlo, copiarlo y distribuirlo (a cualquiera y a cualquier lugar), estudiarlo, cambiarlo y mejorarlo, sin tener que pedir ni pagar permisos al desarrollador original ni a ninguna otra entidad específica. La distribución de las copias puede ser con o sin modificaciones propias, y atención, puede ser gratis ¡o no!: el "software libre" es un asunto de libertad, no de precio.

Para que un programa sea considerado libre a efectos legales ha de someterse a algún tipo de licencia de distribución, entre las cuales se encuentran la licencia GPL (General Public License), o la LGPL, entre otras. El tema de las diferentes licencias es un poco complicado: hay muchas y con muchas cláusulas. Para saber más sobre este tema, se puede consultar <http://www.opensource.org/licenses/category>, donde está disponible el texto oficial original de las licencias más importantes. Ejemplos de software libre hay muchos: el kernel Linux, el navegador Firefox, la suite ofimática LibreOffice, el reproductor multimedia VLC, etc.

El software Arduino es software libre porque está disponible públicamente (en <https://github.com/arduino/Arduino>) con una combinación de la licencia GPL (para el entorno visual de programación propiamente dicho) y la licencia LGPL (para los códigos fuente de gestión y control del microcontrolador a nivel más interno). Una consecuencia de esto es que cualquier persona que quiera (y sepa), puede formar parte del desarrollo del software Arduino y contribuir así a mejorarlo, aportando nuevas características (en <https://github.com/arduino/Arduino/pulls>), sugiriendo ideas que añadirían nuevas funcionalidades o perfeccionarían las ya existentes (en <https://groups.google.com/a/arduino.cc/forum/#forum/developers>), compartiendo soluciones a las carencias y los errores detectados (en <https://github.com/arduino/Arduino/issues>), etc. Esta manera de funcionar provoca la creación espontánea de una comunidad de personas que colaboran mutuamente a través de Internet, y consigue que el software Arduino evolucione según lo que la propia comunidad decida. Esto va mucho más allá de la simple cuestión de si el software Arduino es gratis o no, porque el usuario deja de ser un sujeto pasivo para pasar a ser (si quiere) un sujeto activo y partícipe del proyecto.

¿QUÉ QUIERE DECIR QUE ARDUINO SEA "HARDWARE LIBRE"?

El hardware libre (también llamado "open-source" o "de fuente abierta") comparte muchos de los principios y metodologías del software libre. En particular, el hardware libre permite que la gente pueda estudiarlo para entender su

CAPÍTULO 2: HARDWARE GENUINO

funcionamiento, modificarlo, reutilizarlo, mejorarlo y compartir dichos cambios. Para conseguir esto, la comunidad ha de poder tener acceso a los ficheros esquemáticos del diseño del hardware en cuestión (que son ficheros de tipo CAD). Estos ficheros detallan toda la información necesaria para que cualquier persona con los materiales, herramientas y conocimientos adecuados pueda reconstruir dicho hardware por su cuenta sin problemas, ya que consultando estos ficheros se puede conocer qué componentes individuales integran el hardware y qué interconexiones existen entre cada uno de ellos.

Las placas Arduino son todas hardware libre porque sus ficheros esquemáticos están disponibles para descargar (en sus respectivas páginas dentro de la sección "Productos" de la web <http://www.arduino.cc>) con la licencia Creative Commons Attribution Share-Alike (<http://es.creativecommons.org/licencia>), la cual es una licencia libre que permite realizar trabajos derivados tanto personales como comerciales (siempre que estos den crédito a Arduino y publiquen sus diseños bajo la misma licencia). Así pues, uno mismo se puede construir su propia placa Arduino "en casa" si dispone de los componentes y las herramientas necesarias. No obstante, lo más normal es comprarlas de un distribuidor ya preensambladas y listas para usar; en ese caso, lógicamente, la placa Arduino, aunque sea libre, no puede ser gratuita, ya que es un objeto físico y su fabricación cuesta dinero.

A diferencia del mundo del software libre, donde el ecosistema de licencias libres es rico y variado, en el ámbito del hardware todavía no existen prácticamente licencias específicas de hardware libre, ya que el concepto de "hardware libre" es relativamente nuevo. De hecho, hasta hace poco no existía un consenso generalizado en su definición. Para empezar a remediar esta situación, en el año 2010 surgió el proyecto OSHD (<http://www.oshwa.org/definition>), el cual pretende establecer una colección de principios que ayuden a identificar como "hardware libre" un producto físico. OSHD no es una licencia (es decir, un contrato legal), sino una declaración de intenciones (es decir, una lista general de normas y de características) aplicable a cualquier artefacto físico para que pueda ser considerado libre. El objetivo de la OSHD (en cuya redacción ha participado gente relacionada con el proyecto Arduino, entre otros) es ofrecer un marco de referencia donde se respete por un lado la libertad de los creadores para controlar su propia tecnología y al mismo tiempo se establezcan los mecanismos adecuados para compartir el conocimiento y fomentar el comercio a través del intercambio abierto de diseños. En otras palabras: mostrar que puede existir una alternativa a las patentes de hardware que tampoco sea necesariamente el dominio público. El proyecto OSHD abre, pues, un camino que crea precedentes legales para facilitar el siguiente paso lógico del proceso: la creación de licencias libres de hardware.

EL MUNDO GENUINO-ARDUINO

El objetivo del hardware libre es, por tanto, facilitar y acercar la electrónica, la robótica y en definitiva la tecnología actual a la gente, no de una manera pasiva, meramente consumista, sino de manera activa, involucrando al usuario final para que entienda y obtenga más valor de la tecnología actual e incluso ofreciéndole la posibilidad de participar en la creación de futuras tecnologías: el hardware abierto significa tener la posibilidad de mirar qué es lo que hay dentro de las cosas, y que eso sea éticamente correcto. Permite, en definitiva, mejorar la educación de las personas. Por eso el concepto de software y hardware libre es tan importante, no solo para el mundo de la informática y de la electrónica, sino para la vida en general.

¿POR QUÉ ELEGIR ARDUINO?

Existen muchas otras placas de diferentes fabricantes que, aunque incorporan diferentes modelos de microcontroladores, son comparables y ofrecen una funcionalidad más o menos similar a la de las placas Arduino. Todas ellas también vienen acompañadas de un entorno de desarrollo agradable y cómodo y de un lenguaje de programación sencillo y completo. No obstante, la plataforma Arduino (hardware + software) ofrece una serie de ventajas:

Arduino es libre y extensible: esto quiere decir que, tal como ya hemos comentado en párrafos anteriores, cualquiera que desee ampliar y mejorar tanto el diseño hardware de las placas como el entorno de desarrollo software y el propio lenguaje de programación, puede hacerlo sin problemas. Esto permite que exista un rico "ecosistema" de extensiones, tanto de variantes de placas no oficiales como de librerías software de terceros, que pueden adaptarse mejor a nuestras necesidades concretas.

NOTA: No será hasta el próximo capítulo donde estudiaremos qué son y para qué sirven las "librerías", pero de momento bastará con entenderlas como extensiones del lenguaje Arduino estándar que amplían la funcionalidad básica de este.

Arduino tiene una gran comunidad: muchas personas lo utilizan, enriquecen la documentación y comparten continuamente sus ideas.

Su entorno de programación es multiplataforma: se puede instalar y ejecutar en sistemas Windows, OS X y Linux. Esto no ocurre con el software de muchas otras placas.

Su entorno y el lenguaje de programación son simples y claros: son muy fáciles de aprender y de utilizar, a la vez que flexibles y completos para que los usuarios avanzados puedan aprovechar y exprimir todas las posibilidades del hardware. Además, están bien documentados, con ejemplos detallados y gran cantidad de proyectos publicados en diferentes formatos.

Las placas Arduino son reutilizables y versátiles: reutilizables porque se puede aprovechar la misma placa para varios proyectos (ya que es muy fácil de desconectarla, reconectarla y reprogramarla), y versátiles porque las placas Arduino proveen varios tipos diferentes de entradas y salidas de datos, los cuales permiten capturar información de sensores y enviar señales a actuadores de múltiples formas.

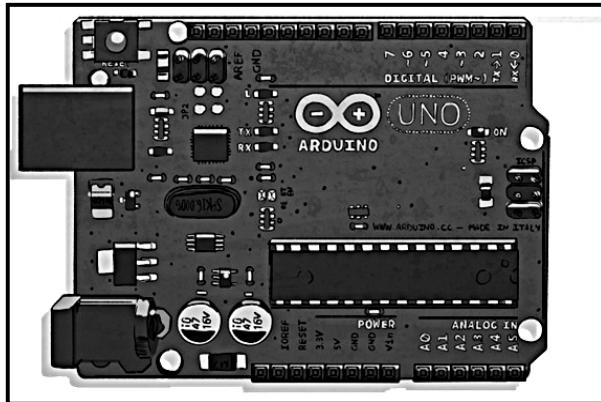
EL MICRO DE LAS PLACAS ARDUINO (y del modelo UNO en particular)

Ya se ha comentado anteriormente que existen varios tipos de placas Arduino, cada una con características específicas que hay que conocer para poder elegir el modelo que más nos convenga según el caso. No obstante, existe un modelo "estándar" de placa, que es el más utilizado con diferencia y que es por donde la mayoría de gente toma su primer contacto con el ecosistema Arduino: la placa UNO. Es por esto que este modelo será el que utilizaremos como referencia en este libro, indicándose oportunamente, eso sí, cualquier diferencia relevante que exista con otros modelos.

A continuación pasaremos a detallar las características técnicas más importantes de la placa UNO, extrapolando los conceptos al resto de placas del ecosistema Arduino cuando sea necesario.

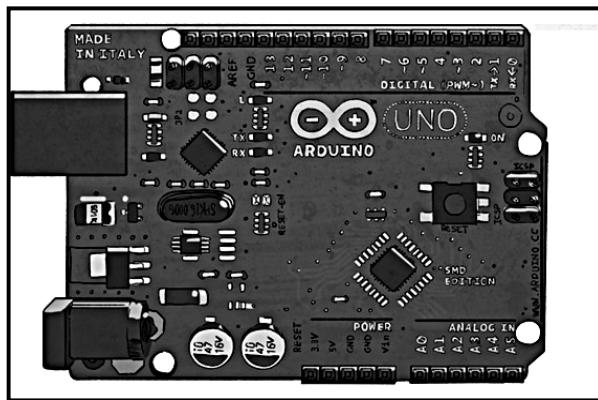
El encapsulado del microcontrolador

Desde que apareció en 2010, la placa UNO ha sufrido tres revisiones, por lo que el modelo actual se suele llamar UNO Rev3 o simplemente UNO R3, y tiene una apariencia similar a la siguiente:



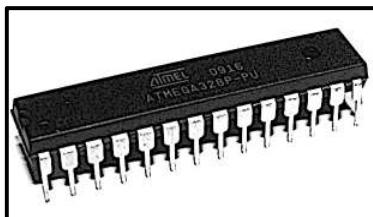
EL MUNDO GENUINO-ARDUINO

La imagen anterior muestra la placa Arduino UNO en su variante convencional. La imagen siguiente muestra, en cambio, la variante llamada Arduino UNO "SMD".



La única diferencia entre ambas placas es el encapsulado físico del microcontrolador incorporado: ambas tienen el mismo modelo, pero la placa convencional lo lleva montado en formato DIP ("Dual In-line Package") y la placa "SMD" lo lleva en formato TQFP ("Thin Quad Flat Pack", una variante de la familia de encapsulados de tipo SMD –"Surface Mount Device"–). En la práctica, esto no nos debería importar en absoluto, a no ser que deseáramos separar y reutilizar el microcontrolador de nuestra placa en otros montajes (como, por ejemplo, en la construcción de una placa Arduino completa sobre una breadboard a partir de componentes individuales, por decir una posibilidad); en ese caso, deberíamos optar por el formato DIP (formato que, dicho sea de paso, solamente aparece en la placa UNO).

DIP



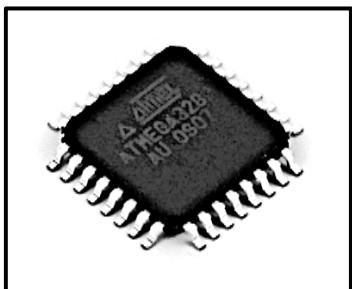
En general, el formato DIP (que en la primera de las imágenes anteriores se muestra como un gran rectángulo en el centro-inferior-derecha de la placa Arduino) se caracteriza por disponer de varias patillas metálicas (cuyo número es diferente dependiendo del chip en cuestión: 8, 14, 16, 24, 28, 36, 64...) ubicadas en ambos costados del encapsulado y espaciadas entre sí 0,1 pulgadas (2,54mm) de manera que pueda ser colocado, como es el caso de la placa UNO, sobre un zócalo soldado a la PCB que permite la fácil sustitución del chip o bien sobre una breadboard, ya que los agujeros de estas tienen esa misma separación y la anchura del encapsulado es la idónea para

el correcto funcionamiento del dispositivo.

poder conectar cada fila de patillas a un lado diferente del área central de la breadboard (y así aislarlas eléctricamente, algo imprescindible para su correcto funcionamiento). También es posible soldar directamente las patillas de un chip DIP a una perfboard o PCB usando una técnica muy sencilla llamada "through-hole" (THT), la cual básicamente consiste –tal como indica su nombre en inglés, "a través del agujero"– en atravesar con cada patilla un orificio de la placa en cuestión y seguidamente soldar cada una de estas patillas a la cara inferior de dicha perfboard/PCB (cortando opcionalmente, una vez realizada la soldadura, los milímetros de patilla que puedan sobresalir por debajo).

Cada patilla de cualquier chip DIP tiene una función específica (alimentación, tierra, entrada/salida...), y aunque esta sea diferente dependiendo de qué chip en concreto estemos tratando, en cualquier caso siempre nos será útil identificar cada patilla individualmente. Para ello existe un convenio entre fabricantes de marcar la patilla nº1 con una muesca en un lateral del encapsulado (y/o con un punto grabado junto a ella); si colocamos entonces la muesca a nuestra izquierda, la patilla nº1 será la situada en el extremo izquierdo de la fila inferior y el resto de patillas se numerarán secuencialmente (2, 3, 4...) en dirección contraria a las agujas del reloj.

SMD



Los chips cuyo encapsulado pertenece a la familia de formatos de tipo SMD, por el contrario, no pueden ser soldados a una perfboard mediante la técnica THT ni tampoco pueden ser acoplados a una breadboard porque no poseen patillas, sino pequeñas (a veces, diminutas) pestañas metálicas diseñadas para ser soldadas a PCBs usando alguna técnica de las llamadas genéricamente "surface-mount technology" (SMT), las cuales comparten la característica –tal como indica su nombre en inglés, "montaje superficial"– de obligar a realizar la soldadura de las pestañas sobre la superficie de la PCB (más en concreto, sobre las vías de cobre preexistentes en la capa exterior de la PCB y especialmente ubicadas para cuadrar con la disposición de las pestañas, a modo de minúsculos "zócalos").

Los chips SMD son más pequeños, más baratos y permiten ser soldados a ambos lados de las placas (permitiendo así una densidad de componentes mucho mayor) pero su manejo es más complicado para los electrónicos aficionados porque se requiere experiencia (e incluso, a veces, instrumentación específica) para realizar las –obligatorias– soldaduras.

EL MUNDO GENUINO-ARDUINO

Afortunadamente, en la mayoría de ocasiones no tendremos que preocuparnos de soldar chips SMD porque los más comunes suelen estar disponibles en el mercado en forma de placas "breakout". Una placa "breakout" simplemente es una placa PCB que lleva soldado un chip SMD (o más) junto con los conectores y la circuitería necesaria para permitir enchufar a ella dispositivos externos de una forma cómoda y rápida (generalmente a través de pines macho, ya soldados de fábrica o bien fácilmente soldables mediante THT). Por tanto, cuando necesitemos en nuestros proyectos algún chip SMD en particular, recurrirremos a alguna placa breakout que lo incorpore y entonces simplemente deberemos utilizar los conectores ofrecidos por ella para poner el chip en comunicación con el resto del circuito. De hecho, podríamos considerar la placa Arduino UNO "SMD" un caso extremo de placa breakout para el microcontrolador que lleva incorporado (el cual se muestra, en la segunda imagen de este apartado, como un pequeño cuadrado ubicado en diagonal en el centro-inferior-derecha de la placa).

El formato SMD concreto del microcontrolador presente en la placa Arduino UNO "SMD" (así como en la mayoría de las otras placas Arduino oficiales) es, recordemos, uno llamado TQFP. Este formato es solo uno más de los diversos tipos de encapsulados SMD que aparecen agrupados bajo la denominación común de QFP ("Quad Flat Pack"). Los formatos QFP comparten la característica de ofrecer un encapsulado cuadrado y plano (el cual tiene en su parte inferior un cojinete térmico que ha de ser soldado a la placa también) y disponer sus pestañas metálicas (generalmente en forma de "ala de gaviota") a lo largo de los cuatro lados de ese encapsulado. El número total de pestañas variará según el chip concreto que estemos tratando (puede ir desde la treintena hasta las centenas) pero, al igual que ocurría con el formato DIP, cada una de las pestañas presentes deberá estar identificada individualmente (para ello también se suele emplear una muesca señalando la nº 1) al tener asociada una función específica. En general, dentro de los formatos QFP podemos encontrar diferentes variantes según el tamaño del encapsulado y el tamaño y espaciado de las pestañas (del orden de las décimas de milímetros); así, tenemos el "Thin" QFP (TQFP), el "Very Thin" QFP (VQFP), el "Low Profile" QFP (LQFP), etc.

Otra subcategoría del formato SMD que podemos encontrar en algunos de los microcontroladores presentes en placas Arduino oficiales diferentes de la UNO (concretamente, en los de los modelos Zero y Due) es el llamado QFN ("Quad Flat No Leads"), también conocido como MLF ("Micro Lead Frame"). Los encapsulados QFN comparten la característica de ser, como los QFP, cuadrados y planos pero, a diferencia de estos, no ofrecen pestañas sobresalientes sino que sus conexiones son pequeñas tiras metálicas ubicadas en la cara inferior del encapsulado. Igualmente, en general podemos hablar del formato "Thin" QFN (TQFN), el "Very Thin" QFN (VQFN),

etcétera (según el tamaño del chip) o, si las tiras aparecen en ambas caras del encapsulado, del formato DFN ("Dual No Lead") o "Thin" DFN (TDFN).

Existen muchos otros formatos de encapsulados más allá de los utilizados en los microcontroladores de las placas Arduino (es decir, más allá de DIP, TQFP y –en menor medida– QFN). Por ejemplo, un formato también muy común en diferentes chips es el llamado SOIC ("Small Outline IC"), que aun siendo de tipo SMD recuerda bastante al DIP porque su diseño básicamente consiste en doblar las patillas "hacia afuera" reduciendo, eso sí, su tamaño y su espaciado. Otro formato popular es el llamado SSOP ("Shrink Small Outline Package"), versión de SOIC más pequeña, así como los formatos –también similares pero de dimensiones aún más pequeñas– TSOP ("Thin Small Outline Package") y TSSOP ("Thin Shrink Small Outline Package"). Por otro lado, en chips certamente complejos nos podemos encontrar con el formato BGA ("Ball Grid Array"), consistente en un encapsulado realmente minúsculo donde las conexiones se realizan a través de pequeñas bolitas metálicas dispuestas en forma de rejilla 2D en la cara inferior del chip. Y existen muchos formatos más que no hemos mencionado; si el lector deseara profundizar en este tema, recomiendo la consulta del artículo <https://goo.gl/RKTtUk>. En cualquier caso, toda esta variedad en los formatos no es un antojo: se debe a las diferentes necesidades existentes respecto la disponibilidad de espacio físico y la disposición de los conectores en cada diseño de PCB.

Finalmente, hay que saber que otros componentes simples que no son circuitos integrados (tales como resistencias, condensadores, diodos, fusibles, etc.) también pueden estar encapsulados en formato SMD para optimizar al máximo el espacio físico ocupado. En estos casos, la variedad de formas y tamaños, aunque estandarizada, es inmensa, y se sale de los objetivos de este libro profundizar en este tema. Baste decir que muchos de estos encapsulados se suelen distinguir por un código de cuatro dígitos, los dos primeros de los cuales indican la longitud del componente y los dos últimos su anchura, en centésimas de pulgadas. Por ejemplo, el encapsulado (por otra parte muy común) "0603" indicaría que el componente en cuestión tiene un tamaño de 0,06" x 0,03". Otros encapsulados comunes son el "0805" y el "0402".

El modelo del microcontrolador

El microcontrolador que lleva la placa Arduino UNO es el modelo ATmega328P, diseñado y fabricado por la marca Atmel (<http://www.atmel.com>). La "P" del final significa que este chip incorpora la tecnología "Picopower" (propiedad de Atmel), la cual permite un consumo eléctrico sensiblemente menor comparándolo

EL MUNDO GENUINO-ARDUINO

con el modelo equivalente sin "Picopower", el Atmega328 (sin la "P"). De todas formas, aunque el ATmega328P pueda trabajar a un voltaje menor y consumir menos corriente que el Atmega328 (especialmente en los modos de hibernación), ambos modelos son funcionalmente idénticos.

Arquitectura AVR

El ATmega328P tiene una arquitectura interna (es decir, un diseño y una estructura funcional) de tipo AVR; más concretamente, pertenece a la subfamilia de microcontroladores "megaAVR". Otras subfamilias de la arquitectura AVR son la "tinyAVR" (cuyos microcontroladores son más limitados de memoria y se identifican con el nombre de ATtiny) y la "XMEGA" (cuyos microcontroladores son más capaces y se identifican con el nombre de ATxmega) pero estas dos no las estudiaremos porque la gran mayoría de placas Arduino contienen un microcontrolador de tipo "megaAVR" (de hecho, tal como iremos viendo, solo son tres los microcontroladores utilizados en prácticamente todo el ecosistema Arduino: el propio ATmega328P, el ATmega32U4 y el ATmega2560, y los tres son "megaAVR") a excepción de la placa Arduino Gemma (que utiliza el microcontrolador ATTiny85, de tipo "tinyAVR") y las placas Arduino Zero y Arduino Due (que utilizan otro tipo completamente diferente de microcontrolador del cual hablaremos enseguida).

La arquitectura AVR está desarrollada y fabricada enteramente por Atmel y es "competencia" de otras arquitecturas como la PIC (del fabricante Microchip) o la MSP430 (del fabricante Texas Instruments), por mencionar un par.

Si se desea saber más sobre la arquitectura AVR y los modelos y características que ofrecen los microcontroladores construidos de esta forma, no hay nada mejor como consultar la web de la propia empresa fabricante: <http://www.atmel.com/products/microcontrollers/avr/default.aspx>. Si lo que se desea, concretamente, es conocer la subfamilia "megaAVR", su documentación se encuentra en <http://www.atmel.com/products/microcontrollers/avr/megaAVR.aspx>. La documentación específica sobre el microcontrolador ATmega328P se puede consultar en <http://www.atmel.com/devices/ATMEGA328P.aspx>, página desde donde se puede descargar (aunque para los proyectos de este libro no será necesario) el "datasheet" de este microcontrolador (el enlace directo de descarga es <http://www.atmel.com/Images/doc8161.pdf>).

Arquitectura ARM

Tal como hemos avanzado en párrafos anteriores, existen dos placas Arduino (concretamente, los modelos Zero y Due) cuyo microcontrolador no es de arquitectura AVR sino de otra arquitectura interna diferente llamada ARM. Al igual que ocurría con AVR, en la arquitectura ARM existen diferentes subfamilias, cada una

CAPÍTULO 2: HARDWARE GENUINO

con diferentes características. Concretamente, el microcontrolador que incorpora la placa Arduino Zero (el SAM-D21) pertenece a la subfamilia "Cortex-M0+" y el que viene en la placa Arduino Due (el SAM3X-8E) pertenece a la subfamilia "Cortex-M3", que es algo más versátil que la anterior ya que pertenece a la generación ARMv7-M (mientras que la subfamilia "Cortex-M0+" pertenece a la generación ARMv6-M). En realidad, el número de subfamilias dentro de la arquitectura ARM es amplísimo, lo que conlleva una gran disparidad de funcionalidades y capacidades: hoy en día podemos encontrar chips ARM dentro de prácticamente cualquier tipo de aparato electrónico: computadores, teléfonos móviles, satélites, lavadoras, etc., etc., etc.

La arquitectura ARM (en sus múltiples subfamilias) está diseñada por la empresa ARM Holdings (<http://www.arm.com>) pero la fabricación física de los chips que implementan dicha arquitectura corre a cargo de diferentes marcas que han conseguido de "ARM Holdings" la licencia necesaria para ello. Esto ayuda a que la presencia de chips ARM se extienda por el mercado, independientemente de la empresa concreta que los haya fabricado. De hecho, los microcontroladores de las placas Zero y Due son de arquitectura ARM pero ambos están fabricados, como los AVR, por Atmel (la cual, por motivos de estrategia comercial, llama "SAM" –de "SMART ARM-based MCU"– a todos sus chips de arquitectura ARM).

Si se desea saber más sobre los modelos y características que ofrecen los microcontroladores ARM construidos por Atmel (los "SAM"), se puede obtener una visión global entrando en <http://www.atmel.com/products/microcontrollers/arm>. De ahí o bien podemos ir a <http://www.atmel.com/products/microcontrollers/arm/sam-d.aspx> para obtener la documentación del conjunto de chips de tipo Cortex-M0+ llamados genéricamente "SAM-D", entre los cuales se encuentra el chip SAM-D21(G18) presente en la placa Arduino Zero (su documentación específica se halla en <http://www.atmel.com/devices/ATSAMD21G18A.aspx>), o bien podemos ir a <http://www.atmel.com/products/microcontrollers/arm/sam3x.aspx> para obtener la documentación del conjunto de chips de tipo Cortex-M3 llamados genéricamente "SAM3X", entre los cuales se encuentra el chip SAM3X-8E presente en la placa Arduino Due (y cuya documentación específica se halla en <http://www.atmel.com/devices/SAM3X8E.aspx>

Breve nota sobre AVR vs. ARM (y x86)

La gran diferencia entre los microcontroladores de tipo AVR y los de tipo ARM (al menos entre los utilizados en las placas Arduino) es su respectivo "tamaño de registro": el de los AVR es de 8 bits y el de los ARM es de 32 bits.

NOTA: Un bit es la unidad mínima de información que puede manejar un microcontrolador, pudiendo valer 0 o 1. La siguiente "breve nota" profundiza en el concepto.

Este tamaño define la cantidad de datos que el microcontrolador es capaz de manejar de forma unitaria. Esto es, es el tamaño del bloque de bits utilizado por defecto en su funcionamiento global: ya sea para leer o escribir datos en direcciones de memoria, o para realizar cálculos con operandos y obtener un resultado, o para enviar o recibir datos al/del exterior, etc., el microcontrolador utiliza siempre su tamaño de registro como tamaño de su "átomo de información" a manipular. Esto significa que los microcontroladores de 32 bits son capaces de procesar de golpe pedazos de bits cuatro veces mayores que los de 8 bits, lo que implica que los primeros son mucho más rápidos y capaces que los segundos.

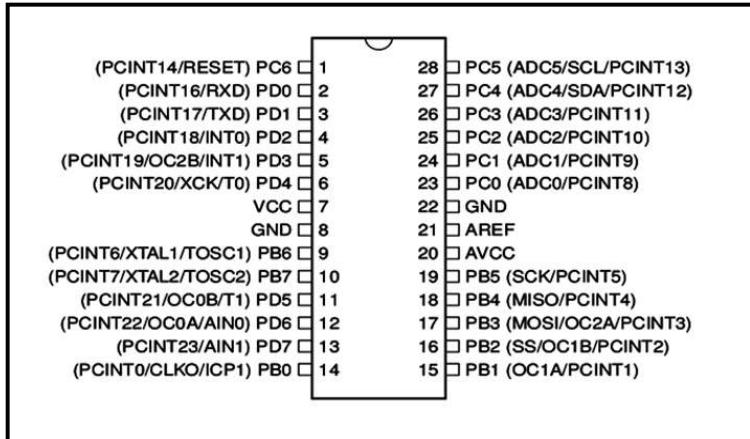
Por otro lado, otro factor a tener en cuenta es que los chips ARM realizan un menor –y más óptimo– consumo energético que los AVR. No obstante, para la gran mayoría de proyectos, con un microcontrolador de 8 bits (como el ATmega328P) ya nos será más que suficiente.

Comentaremos finalmente que en 2016 apareció la placa "Arduino 101", el primer –y único hasta ahora– modelo de placa Arduino con un chip integrado de arquitectura "x86": el llamado Intel Quark. La arquitectura x86 es también de 32 bits pero, en este caso, está desarrollada en exclusiva por el fabricante Intel (<http://www.intel.com>).

El chip ATmega328P

Volviendo al ATmega328P de la placa UNO (recordemos, de arquitectura AVR, concretamente "megaAVR"), algo que nos puede venir muy bien es identificar cada una de sus patillas (llamadas también "pines") porque, tal como ya hemos dicho anteriormente, en general cada pin de cualquier chip tiene una determinada función específica (alimentación, tierra, entrada/salida con el mundo exterior, etc.) y el ATmega328P no es diferente en este sentido.

La disposición de pines del ATmega328P se puede conocer consultando su "datasheet" (descargable de la web de Atmel, como ya sabemos). La figura siguiente, extraída de dicho documento, muestra concretamente la disposición en el encapsulado de tipo DIP (el circulito que aparece en la parte superior de la figura indica el lugar donde existe una muesca en el encapsulado real, de manera que así sea fácil distinguir la orientación de los pines):



Observando la imagen se puede saber qué pin es el que recibe la alimentación eléctrica (señalado como "VCC"), qué dos pines están conectados a tierra (los señalados como "GND"), qué pines son los de E/S digitales (señalados como PBx, PCxo PDx) y la existencia de otros pines más específicos como el AVCC (donde se recibe la alimentación específica para el convertidor analógico-digital interno del chip) o el AREF (donde se recibe la referencia analógica para dicho convertidor –esto lo estudiaremos más adelante–). También se puede observar que junto al nombre de los pines de E/S digitales se indica entre paréntesis las funciones especializadas que cada uno de ellos tiene en particular (además de su función genérica de entrada/salida digital); algunas de estas funciones específicas las iremos estudiando a lo largo del libro, como por ejemplo la función de "reset" del microcontrolador (pin PC6), o la comunicación con el exterior usando el protocolo serie vía UART (pines PD0 y PD1, marcados como "RXD" y "TXD"), vía SPI (pines PB2, PB3, PB4 y PB5, marcados como "SS", "MOSI", "MISO" y "SCK") o vía I²C (pines PC4 y PC5, marcados como "SDA" y "SCL"), o el uso de interrupciones (pines PD2 y PD3, marcados como "INT1" y "INT2"), o el de las salidas PWM (pines PD3, PD5, PD6, PB1, PB2 y PB3, marcados como "OC0x", "OC1x" y "OC2x"), o el de las entradas analógicas (pines PC0, PC1, PC2, PC3, PC4 y PC5, marcados como "ADCx"), o la conexión con el oscilador de cristal externo (pines PB6 y PB7, marcados como "XTALx"), etc.

Las memorias del microcontrolador

Otra cosa que hay que saber de los microcontroladores son los tipos y las cantidades de memoria que alojan en su interior. Tanto en el caso de los microcontroladores AVR como en el de los ARM tenemos:

EL MUNDO GENUINO-ARDUINO

Memoria Flash: memoria persistente donde se almacena permanentemente el programa que ejecuta el microcontrolador (hasta una eventual nueva reescritura, la cual, en cualquier caso, nunca podrá realizarse mientras se esté ejecutando el programa actual). El ATmega328P en concreto tiene una capacidad de 32KB.

Breve nota sobre las unidades de medida de la información

Es buen momento para repasar cómo se mide la cantidad de datos almacenados en una memoria (o transferidos por algún canal). El sistema de medición de la información es el siguiente:

1 bit	Unidad mínima (puede valer 0 o 1)
1 byte	Grupo de 8 bits
1 kilobyte (a veces escrito como KB)	Grupo de 1024 bytes (es decir, 8192 bits)
1 megabyte (MB)	Grupo de 1024 KB (es decir, 1048576 bytes)
1 gigabyte (GB)	Grupo de 1024 MB

Observar que los múltiplos "kilo", "mega" y "giga" hacen referencia a agrupaciones de 1024 (2^{10}) elementos, en vez de lo que suele ocurrir con el resto de unidades de medida del mundo físico, donde son agrupaciones de 1000 elementos. Esta discrepancia se debe a la naturaleza intrínsecamente binaria de los componentes electrónicos.

Observar también que no es lo mismo Kbit ("kilobit") que KB ("kilobyte"). En el primer caso estaríamos hablando de 1024 bits, y en el segundo de 1024 bytes (es decir, 8192 bits, ocho veces más).

En los microcontroladores que vienen incluidos en las placas Arduino no podemos usar toda la capacidad de su memoria Flash porque dentro de ella existe siempre una zona reservada (el llamado "bootloader block") que está ocupada por un código preprogramado de fábrica (el llamado "bootloader" o "gestor de arranque"), el cual nos permite usar la placa Arduino de una forma sencilla y cómoda sin tener que conocer las interioridades electrónicas más avanzadas del microcontrolador. Los microcontroladores que podemos adquirir de forma individual (que en la práctica, solo serán los Atmega328P en formato DIP) normalmente no incluyen de fábrica ningún bootloader –esto es, ninguna "preconfiguración"–, por lo que aunque entonces sí nos ofrecen toda su memoria Flash para poder alojar en ella código propio, no podemos esperar conectarlos a una placa Arduino y que funcionen sin más, ya que les faltará tener grabada precisamente esa "preconfiguración" que consigue que una placa Arduino sea tan sencilla de manejar. El tamaño del

CAPÍTULO 2: HARDWARE GENUINO

"bootloader block" depende del modelo de placa Arduino donde se encuentre el microcontrolador en cuestión; en el caso, por ejemplo, del ATmega328P incorporado en la placa UNO, es de 512 bytes (valor que, como podemos ver, no llega al 2% del tamaño total de su memoria Flash). De los gestores de arranque, de su uso y su importancia hablaremos en un próximo apartado.

En realidad, en las memorias Flash no solamente podemos almacenar el programa ejecutado por el microcontrolador de la placa (y el "bootloader"), sino que técnicamente es posible alojar allí cualquier tipo de información. Esto significa que si en nuestros proyectos necesitáramos almacenar contenido extra que no cupiera en la memoria Flash ofrecida por el microcontrolador de la placa Arduino utilizada, una opción sería conectar a dicha placa algún módulo específico diseñado para aportar memoria Flash suplementaria. Por ejemplo, el fabricante LcTech-Inc (<http://www.lctech-inc.com/Hardware>) distribuye varios módulos (basados en la familia de chips W25Q de Winbond) que ofrecen diferentes cantidades de memoria de tipo Flash; todos ellos son capaces de comunicarse a través del protocolo SPI (que estudiaremos en un próximo apartado) y, por tanto, son programables mediante la librería oficial "SPI" de Arduino. Adafruit distribuye por su parte uno de estos chips W25Q directamente en formato DIP como producto **nº 1564**.

NOTA: Adafruit también distribuye los productos **nº 1895** (en forma de plaquita breakout conectable vía I²C) y **nº 1897** (en forma de plaquita breakout conectable vía SPI) que aportan respectivamente hasta 64 Kbits y 32 Kbits de otro tipo de memoria permanente similar a la Flash (pero más rápida) llamada FRAM. Ambos módulos pueden ser programados mediante una librería propia de Adafruit descargable de su web.

Otra opción para añadir más cantidad de memoria Flash a nuestra placa Arduino es adquirir tarjetas de memoria de tipo SD ("Secure Digital") y comunicarlas por medio de un circuito específico a la placa en cuestión. Las memorias SD son memorias Flash que están encapsuladas de una forma concreta y son accesibles a través del protocolo SPI y gestionables mediante la librería oficial "SD" del lenguaje Arduino (que estudiaremos en el capítulo 5); su uso está muy extendido en cámaras digitales de foto/vídeo y en teléfonos móviles de última generación porque ofrecen muchísima capacidad (varios gigabytes) a un precio barato.

NOTA: Hay que tener en cuenta que si lo que quisiéramos almacenar en la tarjeta SD no fueran simplemente datos sino que fuera precisamente el código de nuestro programa, entonces sería necesario previamente cambiar el "bootloader" del microcontrolador por otro diferente capaz de localizar correctamente dicho programa en su nueva ubicación (ya que el "bootloader" que viene por defecto en las placas Arduino está configurado para reconocer solamente el código alojado en la memoria Flash del propio microcontrolador). El proceso de sobrescritura del gestor de arranque será detallado en el apartado sobre ISP de este mismo capítulo.

EL MUNDO GENUINO-ARDUINO

Memoria SRAM: memoria volátil donde se alojan los datos que en ese instante el programa (grabado separadamente en la memoria Flash, recordemos) necesita crear o manipular para su correcto funcionamiento. Estos datos suelen tener un contenido variable a lo largo del tiempo de ejecución del programa y cada uno es de un tipo concreto (es decir, un dato puede contener un valor numérico entero, otro un número decimal, otro un valor de tipo carácter...). También pueden ser cadenas de texto fijas u otros tipos de datos más especiales). Independientemente del tipo de dato, su valor siempre será eliminado cuando se deje de alimentar eléctricamente al microcontrolador. En el caso del ATmega328P esta memoria tiene una capacidad de 2KB.

Una característica a destacar de este tipo de memorias es su gran rapidez a la hora de obtener un determinado dato solicitado (de entre todos los alojados en ella) así como a la hora de realizar la escritura de un nuevo valor: esta velocidad es mucho mayor que en el caso de los otros tipos de memoria.

Si en nuestros proyectos necesitáramos más cantidad de memoria SRAM de la que nos ofrece el microcontrolador de la placa UNO, una opción sería recurrir a otros modelos de placas Arduino que incorporen un microcontrolador más capaz (serán estudiados en posteriores apartados). Si, no obstante, deseáramos continuar utilizando igualmente el modelo UNO, entonces otra opción podría ser adquirir por separado memorias SRAM independientes y conectarlas al microcontrolador utilizando algún protocolo de comunicación conocido por este (el más habitual suele ser, en estos casos, el SPI, del cual hablaremos enseguida). Memorias de este tipo son, por ejemplo, los chips **23LCV1024**, **23LC512** y **23K256** de Microchip (<http://www.microchip.com>), todos ellos disponibles en formato DIP y manipulables mediante la librería oficial "SPI" de Arduino (los dos primeros funcionan a 5V y ofrecen, respectivamente, 1024KBits –128KBytes– y 512KBits –64KBytes– de memoria SRAM extra mientras que el último funciona a 3,3V y ofrece 256Kbits –32KBytes–); también se pueden encontrar incluso shields específicos que aportan este tipo de memoria, como el **SPI RAM Shield** de Tindie/FemToCow. De todas formas, no será necesario llegar a estos extremos en los proyectos de este libro.

Solo en el caso de los microcontroladores AVR tenemos además, un tercer tipo de memoria:

Memoria EEPROM: memoria persistente donde se almacenan datos que se desea que permanezcan grabados una vez apagado el microcontrolador para así poderlos usar posteriormente en siguientes reinicios (por ejemplo, un número de serie único y/o la fecha de fabricación de algún proyecto comercial, o un determinado conjunto de valores numéricos ya precalculados

para ser así usados directamente sin demora en operaciones matemáticas complejas, o el contador de algún evento cuyo valor no pueda ser reseteado por el usuario, etc.). En el caso del ATmega328P esta memoria tiene una capacidad de 1KB, por lo que se puede entender como una tabla de 1024 posiciones ("celdas") de un byte cada una.

Si necesitáramos ampliar la cantidad de memoria EEPROM disponible (o incluir una memoria de este tipo en el caso de usar las placas Zero o Due), siempre podemos adquirir memorias EEPROM independientes y conectarlas al microcontrolador utilizando algún protocolo de comunicación conocido por este (como SPI o I²C, de los cuales hablaremos enseguida). Por ejemplo, el producto **DFR0117** de DFRobot, basado en el chip AT24C256 de Atmel, es una plaquita breakout conectable mediante I²C y programable mediante la librería oficial "Wire" de Arduino (en la página oficial de DFRobot se ofrece un completo tutorial de uso) que aporta 32Kbytes extra; otros módulos similares (es decir, basados también en un chip Atmel de la familia AT24 conectables mediante I²C y programables mediante "Wire" -aunque cada uno incorporando, eso sí, diferentes cantidades de memoria-), son los ofrecidos dentro del apartado "Storage" de la sección "Hardware" del fabricante LcTech-Inc. Sparkfun, por su lado, distribuye como producto **nº 525** el chip 24LC256 de Microchip directamente en formato DIP, el cual también aporta 32Kbytes de memoria y también es conectable vía I²C y programable con la librería "Wire".

Breve nota sobre las diferencias entre memorias Flash y EEPROM

El lector puede estar confundido sobre el porqué de la existencia de dos tipos de memorias si ambas son permanentes (Flash y EEPROM). ¿Qué aporta una que no aporte la otra? En realidad, podemos considerar la memoria Flash como un tipo concreto de memoria EEPROM pero con unas características muy determinadas:

*Una memoria Flash se puede escribir menos veces que una memoria EEPROM típica antes de dejar de funcionar. Concretamente, en una memoria Flash se pueden realizar del orden de hasta 10000 escrituras mientras que en una memoria EEPROM la cantidad máxima ronda las 100000 escrituras (las memorias SRAM no tienen límite).

*Las memorias EEPROM permiten lecturas/escrituras a nivel de byte. Las memorias Flash, en cambio, solo permiten lecturas/escrituras a nivel de bloque, los cuales suelen tener un tamaño entre 64 y 512 bytes (según el modelo). Esto significa que el acceso a los datos (y su eventual reescritura) en memorias EEPROM es mucho más eficiente y rápido.

*Las memorias EEPROM suelen ser bastante más caras que las memorias Flash y es por ello que se suelen comercializar con menos cantidad de almacenamiento.

Los registros del microcontrolador

Los registros son espacios de memoria existentes dentro de la propia CPU de un microcontrolador. Son muy importantes porque tienen varias funciones imprescindibles: sirven para albergar los datos (cargados previamente desde la memoria SRAM o EEPROM) necesarios para la ejecución de las instrucciones previstas próximamente (y así tenerlos perfectamente disponibles en el momento adecuado); sirven también para almacenar temporalmente los resultados de las instrucciones recientemente ejecutadas (por si se necesitan en algún instante posterior) y sirven además para alojar las propias instrucciones que en ese mismo momento estén ejecutándose.

Su tamaño es muy reducido: tan solo tienen capacidad para almacenar unos pocos bits cada uno, pero este factor es una de las características más importantes que definen a cualquier microcontrolador; de hecho, el tamaño de sus registros es lo que (simplificando mucho) hace que identifiquemos un micro como "de 8 bits" o "de 32 bits". En efecto, cuanto mayor sea el número de bits que "quepan" en sus registros, mayores serán sus prestaciones en cómputo y manipulación de datos: es fácil ver (simplificando mucho otra vez) que un microcontrolador con registros el doble de grandes que otro podrá procesar el doble de cantidad de datos y por tanto, trabajar el "doble de rápido" aun funcionando los dos al mismo ritmo.

Dependiendo de la utilidad que vayamos a darle al microcontrolador, será necesario utilizar uno con un tamaño de registros suficiente. Por ejemplo, el control de un electrodoméstico sencillo como una batidora no requiere más que un microcontrolador de 4 u 8 bits. En cambio, el sistema de control electrónico del motor de un coche o el de un sistema de frenos ABS se basan normalmente en un microcontrolador de 16 o 32 bits. Ya vimos en un apartado anterior de este capítulo que todos los microcontroladores de tipo AVR incluidos en las diferentes placas Arduino (y, por tanto, entre otros, el ATmega328P) son de 8 bits, mientras que los dos únicos micros de tipo ARM presentes en el ecosistema Arduino (en concreto, en las placas Zero y Due) y el chip de tipo x86 (presente en la placa 101) son de 32 bits.

La comunicación serie con el exterior

Cuando se desea transmitir un conjunto de datos desde un componente electrónico a otro, se puede hacer de múltiples formas. Una de ellas es estableciendo una comunicación "serie"; en este tipo de comunicación la información es transmitida bit a bit (uno tras otro) por un único canal, enviando por tanto un solo bit en cada momento. Otra manera de transferir datos es mediante la llamada comunicación "paralela", en la cual se envían varios bits simultáneamente, cada uno por un canal separado y sincronizado con el resto.

CAPÍTULO 2: HARDWARE GENUINO

Los microcontroladores, a través de algunos de sus pines de E/S, utilizan sobre todo sistemas de comunicación serie para transmitir y recibir órdenes y datos hacia/desde otros componentes electrónicos. Esto es debido a que en una comunicación serie solo se necesita un único canal (es decir, un único "cable", aunque esto es en teoría porque ya veremos que en la práctica se necesita alguno más), mientras que en una comunicación en paralelo se necesitan varios canales, con el correspondiente incremento de complejidad, tamaño y coste del circuito resultante.

Hay que tener claro, no obstante, que no podemos hablar de un solo tipo de comunicación serie: existen muchos protocolos y estándares diferentes basados todos ellos en la transferencia de información en serie, pero implementando de una forma diferente cada uno los detalles específicos (como el modo de sincronización entre emisor y receptor, la velocidad de transmisión, el tamaño de los paquetes de datos, los mensajes de conexión y desconexión y de dar paso al otro en el intercambio de información, los voltajes utilizados, etc.). De entre el gran número de protocolos de comunicación serie reconocidos por la inmensa variedad de dispositivos electrónicos del mercado, los que nos interesarán conocer a nosotros son los que los microcontroladores presentes en las diferentes placas Arduino (y, en especial, el ATmega328P) son capaces de comprender y por tanto, utilizar para contactar con esa variedad de periféricos. Teniendo esto en cuenta, los métodos de comunicación serie que en concreto estudiaremos en este libro son solo tres: dos de tipo síncrono (los protocolos I²C y SPI) y uno de tipo asíncrono (denominado muchas veces simplemente "serie" o también "UART" debido al nombre genérico del chip que lo implementa: "Universal Asynchronous Receiver/Transmitter").

Que la comunicación serie sea "síncrona" o "asíncrona" depende de si emisor y receptor transfieren información de forma sincronizada o no. En el primer caso ha de existir una señal periódica compartida por ambos extremos (la cual debe ser transmitida por un cable dedicado en exclusiva a ello a parte del (o los) cable/s empleado/s para la transferencia de datos) ejerciendo de "metrónomo" para marcar las pulsaciones que se encargarán de señalar (tanto a un extremo como al otro) los momentos puntuales en los que pueden enviar o recibir datos. En el caso de la comunicación serie asíncrona no existe ningún "metrónomo" compartido por los extremos, por lo que siempre será necesario que estos acuerden, al inicio de cada conexión, una determinada velocidad de transferencia (medida en "baudios") soportada y reconocida por ambos, además de implementar un sistema de aviso de llegada y/o final de mensaje (en la cabecera y cola del mismo, respectivamente) e intercambiar varios bits extra de control más durante la transmisión/recepción para mantener un reconocimiento correcto de la información transferida entre ellos.

Comunicación asíncrona

Todos los microcontroladores de las placas Arduino (ya funcionen a 5V o a 3,3V) disponen de hardware UART. Los chips UART usan siempre una línea para transmitir datos (llamada normalmente "TX") y otra diferentes para recibirlos (llamada normalmente "RX"). Esto significa que utilizando este método bastan dos cables para comunicar nuestro microcontrolador (a través de los pines-hembra correspondientes de nuestra placa) con el exterior.

NOTA: También hay que tener en cuenta que la tierra de la placa Arduino y del dispositivo exterior con el que esta se comunique ha de ser común a ambos, por lo que muchas veces será necesario unir estos dos extremos también con un tercer cable conectado a un mismo GND, además de la alimentación que cada uno pudiera tener.

Es fácil ver que, al haber dos líneas para los datos la transmisión de información es de tipo "full duplex" (es decir, que la información puede ser transportada en ambos sentidos a la vez).

Comúnmente, los chips UART transmiten conjuntos de 8 bits de datos, uno tras otro, a una determinada velocidad (medida en "baudios"). Cada uno de estos conjuntos (llamados "bits de datos") viene siempre precedido de un bit especial (llamado "bit de inicio") que indica al receptor precisamente el inminente envío de ese byte, y viene siempre seguido por otro bit especial (llamado "bit de parada") que indica el fin del envío de ese conjunto. Opcionalmente, también pueden añadirse a ese conjunto bits adicionales, como ciertos "bits de paridad" (útiles para la detección de errores en la transmisión) o ciertos "bits de control de flujo –flow control–" (útiles para gestionar los posibles "atascos" en el envío de la información). En cualquier caso, tanto la velocidad de transmisión como la cantidad de cada uno de estos bits "especiales" debe ser previamente establecida en los mismos valores para los dos extremos de la comunicación si queremos que esta se pueda realizar correctamente. En este sentido, la velocidad de transmisión del chip UART del microcontrolador de cualquier placa Arduino es un valor que puede ser especificado dentro del código del programa que le cargaremos, pero la cantidad de los diferentes bits especiales es fija y no puede ser cambiada desde el entorno Arduino; afortunadamente, esto no nos será ningún problema porque las placas Arduino emplean unos valores estándar ampliamente utilizados en la gran mayoría de dispositivos; concretamente: 8 "bits de datos", ninguno "bit de paridad", 1 "bit de inicio" y otro de parada, y sin control de flujo (o más abreviadamente, 8-N-1-N).

Comunicación síncrona

Los dos protocolos serie síncronos más comunes en el universo Arduino son:

I²C (Inter-Integrated Circuit, también conocido con el nombre de **TWI** –de "TWo-wire", literalmente "dos cables" en inglés–): es un sistema muy utilizado en la industria principalmente para comunicar circuitos integrados entre sí (a menos de 1m de distancia). Su principal característica es que solo utiliza dos líneas: una (llamada línea "SDA") sirve para transferir los datos (los 0s y los 1s) en un sentido o en el otro y otra (llamada línea "SCL") sirve para enviar la señal de reloj.

NOTA: En realidad también se necesitarían dos líneas más: la de alimentación común y la de tierra común a emisor y receptores, pero estas ya se presuponen existentes en el circuito.

Por "señal de reloj" se entiende una señal binaria de una frecuencia periódica muy precisa que sirve para coordinar y sincronizar los elementos integrantes de una comunicación (es decir, los emisores y receptores) de forma que todos sepan cuándo empieza, cuánto dura y cuándo acaba la transferencia de información. En hojas técnicas y diagramas a la señal de reloj en general se le suele describir como CLK (del inglés "clock").

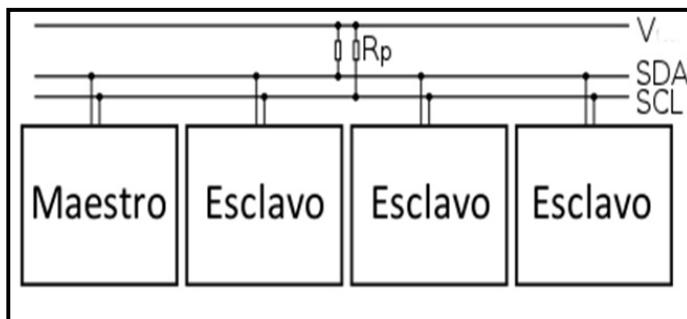
El hecho de que haya una única línea de datos hace que la transmisión de información sea "half duplex" (es decir, que la comunicación solo se pueda establecer en un sentido al mismo tiempo) por lo que en el momento que un dispositivo empiece a recibir un mensaje, tendrá que esperar a que el emisor deje de transmitir para poder responderle. No obstante, a diferencia de lo que ocurría en las conexiones "full duplex" vía UART, donde solo podía haber un dispositivo en cada extremo, las dos líneas empleadas en una comunicación I²C (SDA y SCL) pueden ser compartidas por multitud de dispositivos a la vez, pudiendo existir así múltiples receptores funcionando al mismo tiempo sobre las mismas dos líneas y también múltiples transmisores (aunque en la práctica lo más habitual es que solo haya un único transmisor para todos los receptores conectados). Para que entonces la comunicación I²C dentro de las líneas SDA y SCL compartidas no sea un caos, cada dispositivo conectado a ellas deberá tener una dirección única que lo identifique respecto el resto de dispositivos; esta dirección única es un número que suele venir prefijado de fábrica (hay que consultar, por tanto, el "datasheet" del producto para conocerla) aunque en algunos casos, es posible cambiarlo mediante algún procedimiento mecánico o eléctrico. En el caso concreto de las placas Arduino esa dirección se establece dentro de nuestro código cargado en el microcontrolador.

Un dispositivo que quiera comunicarse a través de I²C, además de tener asignada la dirección identificativa mencionada anteriormente, debe estar previamente configurado como "maestro" o como "esclavo". Un dispositivo maestro

EL MUNDO GENUINO-ARDUINO

es el que inicia la transmisión de datos y además genera la señal de reloj, pero no es necesario que el maestro sea siempre el mismo dispositivo: esta característica se la pueden ir intercambiando ordenadamente los dispositivos que tengan esa capacidad. Hay dispositivos que están diseñados solamente para funcionar como "esclavos" y dispositivos que pueden alternar su comportamiento entre "esclavo" y "maestro", ya sea, otra vez, mediante algún procedimiento mecánico o eléctrico o bien, como es el caso de las placas Arduino, mediante el código apropiado cargado en el microcontrolador.

Finalmente, un detalle muy importante que hay que tener en cuenta es que, tal como muestra el diagrama siguiente, para funcionar correctamente tanto la línea SDA como la SCL necesitan estar conectadas mediante una resistencia "pull-up" a la fuente de alimentación común a todos los dispositivos, la cual puede proveer un voltaje generalmente de 5V o 3,3V (aunque sistemas con otros voltajes pueden ser posibles). En el diagrama no se muestra, pero la línea de tierra (GND) también ha de ser común a todos los dispositivos.



SPI (Serial Peripheral Interface): al igual que el sistema I²C, el sistema de comunicación SPI es un estándar que permite controlar (a cortas distancias) casi cualquier dispositivo electrónico digital que acepte un flujo de bits serie sincronizado (es decir, regulado por un reloj). Igualmente, un dispositivo conectado al bus SPI puede ser "maestro" –en inglés, "master"– o "esclavo" –en inglés, "slave"–, donde el primero es el que inicia la transmisión de datos y además genera la señal de reloj (aunque, como con I²C, con SPI tampoco es necesario que el maestro sea siempre el mismo dispositivo) y el segundo se limita a responder.

La mayor diferencia entre el protocolo SPI y el I²C es que el primero requiere de cuatro líneas ("cables") en vez de dos. Una línea (llamada normalmente "SCK" o "SCLK" -de "Serial Clock"-) envía a todos los dispositivos la señal de reloj generada

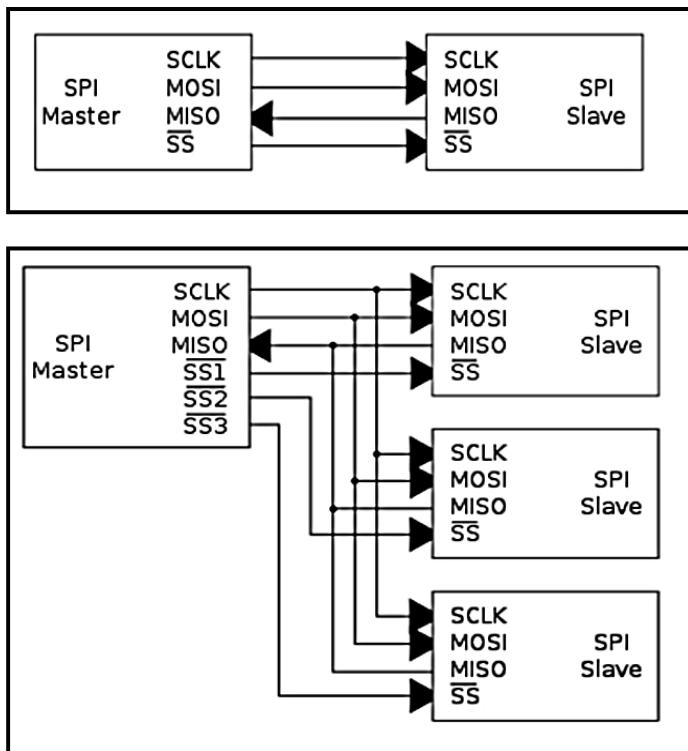
CAPÍTULO 2: HARDWARE GENUINO

por el maestro actual; otra (llamada normalmente "SS" o "CS" –de "Slave Select" o "Chip Select", respectivamente–) es la utilizada por ese maestro para elegir en cada momento con qué dispositivo esclavo se quiere comunicar de entre los varios que puedan estar conectados (ya que solo puede intercambiar datos con un solo esclavo a la vez); otra (llamada normalmente "MOSI" –de "Master Output, Slave Input"–) es la línea utilizada para enviar los datos –0s y 1s– desde el maestro hacia el esclavo elegido; y la otra (llamada normalmente "MISO" –de "Master Input", Slave Output"–) es la utilizada para enviar los datos en sentido contrario: la respuesta de ese esclavo al maestro. Es fácil ver, pues, que al haber dos líneas separadas para los datos la transmisión de información es "full duplex" (es decir, la información puede ser transportada en ambos sentidos a la vez).

En un comunicación SPI siempre ha de haber un dispositivo maestro y uno o más dispositivos esclavos pero, a diferencia de lo que podía ocurrir en el protocolo I²C (aunque no sea muy habitual), la tecnología SPI no permite la existencia de más de un dispositivo maestro. Por otro lado, el lenguaje y entorno Arduino no ofrece la capacidad (a diferencia otra vez del I²C) de configurar ninguna placa como esclavo SPI (pese a que el hardware sí admite esa posibilidad), por lo que obligatoriamente deberemos utilizar cualquiera de ellas como maestro. Esto significa que todos los dispositivos que queramos comunicar a través de SPI a nuestra placa Arduino deberán actuar obligatoriamente como esclavos.

En las siguientes figuras se muestra el esquema de líneas de comunicación existentes entre un maestro y un esclavo y entre un maestro y tres esclavos respectivamente. Se puede observar que, para el caso de la existencia de varios esclavos es necesario utilizar una línea "SS" diferente por cada uno de ellos, ya que esta línea es la que sirve para activar el esclavo concreto con el que en cada momento el maestro desee comunicarse (esto no pasa con las líneas de reloj, "MOSI" y "MISO", que son compartidas por todos los dispositivos). Técnicamente hablando, el esclavo que reciba por su línea SS un valor de voltaje BAJO será el que esté seleccionado en ese momento por el maestro, y los que reciban el valor ALTO no lo estarán (de ahí el subrayado superior que aparece en la figura). A este sistema de seleccionar líneas de comunicación se le llama "activo BAJO".

NOTA: No nos interesaría nunca activar dos esclavos a la vez porque, si así lo hicieramos, el maestro recibiría, por ejemplo, los datos mezclados de ambos esclavos a través de la línea MISO.



Como se puede ver, el protocolo SPI respecto el I²C tiene la desventaja de exigir al microcontrolador dedicar muchos más pines de E/S a la comunicación externa. En cambio, como ventaja podemos destacar que, además de ser más rápido y consumir menos energía que I²C, también es más sencillo de implementar electrónicamente, ya que, entre otras cosas, no requiere del uso de identificadores únicos para los dispositivos (ya que la selección de los interlocutores se realiza, tal como hemos dicho, mediante la simple alternancia de valores ALTO y BAJO en los canales SS).

Tal como se puede observar en la figura que muestra la disposición de pines del microcontrolador ATmega328P (página 99), los pines correspondientes a las líneas I²C SDA y SCL son los números 27 y 28, respectivamente, y los pines correspondientes a las líneas SPI SS, MOSI, MISO y SCK son los números 16, 17, 18 y 19, respectivamente. Si se necesitaran más líneas SS (porque haya más de un dispositivo esclavo conectado en nuestro circuito), se podría utilizar cualquier otro pin de E/S siempre que respete el convenio de poner el valor de su voltaje de salida a BAJO cuando se desee trabajar con el dispositivo esclavo asociado y poner a ALTO el resto de pines SS.

El gestor de arranque ("bootloader") del microcontrolador

Ya hemos comentado en un apartado anterior que en una determinada zona (llamada "bootloader block") de la memoria Flash del microcontrolador incluido en la mayoría de placas Arduino viene pregrabado de fábrica un pequeño programa llamado "bootloader" o "gestor de arranque", que resulta imprescindible para un cómodo y fácil manejo de la placa en cuestión. Este software (también llamado "firmware", porque es un tipo de software que raramente se modifica) ocupa, en la placa Arduino UNO, 512 bytes de espacio de memoria, pero en otros modelos de placas Arduino puede ocupar más.

La función de este firmware es gestionar de forma automática el proceso de grabación en la memoria Flash del programa que queremos que el microcontrolador ejecute. Lógicamente, el bootloader realizará esta grabación más allá del "bootloader block" para no sobrescribirse a sí mismo. Así pues, en otras palabras, un gestor de arranque no es más que un programa pregrabado en una zona de la memoria Flash que sirve para realizar la grabación de otro programa (el nuestro) en el resto de memoria Flash que queda libre.

Más concretamente, el bootloader se encarga de recibir nuestro programa de parte del entorno de desarrollo Arduino (normalmente mediante una transmisión realizada a través de conexión USB desde el computador donde se está ejecutando dicho entorno hasta la placa) para proceder seguidamente a su correcto almacenamiento en la memoria Flash, todo ello de forma automática y sin que nos tengamos que preocupar de las interioridades electrónicas del proceso. Una vez realizado el proceso de grabación, el bootloader termina su ejecución y el microcontrolador se dispone a procesar de inmediato y de forma permanente (mientras esté encendido) las instrucciones recientemente grabadas.

En la placa Arduino UNO, el bootloader siempre se ejecuta durante el primer segundo de cada reinicio. Durante esos instantes, el gestor de arranque se espera a recibir una serie de instrucciones concretas de parte del entorno de desarrollo para interpretarlas y realizar la correspondiente carga de un posible programa. Concretamente, esas instrucciones (y el eventual programa a cargar) son recibidas por el bootloader a través de los pines "RX" y "TX" del chip UART del microcontrolador, los cuales están conectados internamente al zócalo USB de la placa, zócalo que está conectado a su vez (mediante el cable pertinente) al puerto USB de nuestro computador, que es donde el entorno de desarrollo que genera dichas instrucciones las envía (tras haber reiniciado el microcontrolador para que el "bootloader" se ponga en marcha). Si esas instrucciones no llegaran pasado el primer

EL MUNDO GENUINO-ARDUINO

segundo tras el reinicio de la placa, el bootloader terminará su ejecución poniendo en marcha el programa que pudiera haber en ese momento grabado en el resto de la memoria Flash.

Las instrucciones internas de programación de memorias Flash son ligeramente diferentes según el tipo de bootloader que tenga el microcontrolador pero, en el caso de los de arquitectura AVR, casi todas ellas son variantes de las instrucciones oficiales diseñadas por Atmel llamadas en su conjunto "protocolo STK500" (<http://www.atmel.com/tools/STK500.aspx>). Un ejemplo es el bootloader que tiene pregrabado el ATmega328P del Arduino UNO, basado en un firmware libre llamado Optiboot (<https://github.com/Optiboot/optiboot>), el cual mejora la velocidad de grabación de nuestro programa en Flash gracias al uso de instrucciones propias derivadas del "estándar" STK500. Otro ejemplo de bootloader derivado del protocolo STK500 es el bootloader "stk500v2", grabado de fábrica en el microcontrolador de la placa Arduino Mega. En cambio, el bootloader que viene en las placas basadas en el micro ATmega32U4 (como la Micro o la Yún), el cual se llama "Caterina", entiende otro conjunto de instrucciones independiente llamado AVR109 (también oficial de Atmel).

NOTA: Toda esta información se puede obtener consultando el contenido del fichero llamado "boards.txt", existente dentro de la carpeta "hardware/arduino/avr" descargada junto con el entorno de desarrollo de Arduino.

Si adquirimos un microcontrolador ATmega328P por separado, hay que tener en cuenta que no dispondrá del bootloader, por lo que deberemos incorporarle uno nosotros "a mano" para hacer uso de él a partir de entonces, o bien no utilizar nunca ningún bootloader y cargar entonces siempre nuestros programas a la memoria Flash directamente. En ambos casos, el procedimiento requiere, además del entorno software de Arduino, del uso de un aparato específico (en concreto, lo que se llama un "programador ISP" –In System Programmer–) que debemos adquirir aparte. Este aparato se ha de conectar por un lado a nuestro computador y por otro a la placa Arduino, y suple la ausencia de bootloader haciendo de intermediario entre nuestro entorno de desarrollo y la memoria Flash del microcontrolador (hablaremos más en detalle sobre ello en un apartado posterior de este capítulo). Por lo tanto, podemos decir que el gestor de arranque es el elemento que permite programar nuestro Arduino directamente con un simple cable USB sin necesidad de utilizar ningún equipamiento hardware especializado.

Por conveniencia, dentro del paquete instalador del entorno de desarrollo de Arduino (descargable de su web oficial, para más detalles ver el capítulo siguiente), se distribuyen además copias exactas bit a bit de los bootloaders oficiales que vienen grabados en los diferentes microcontroladores Arduino. Estas copias exactas son

ficheros con extensión ".hex" que tienen un formato interno llamado "Intel Hex Format". Para el uso normal de nuestra placa no necesitamos para nada estos ficheros ".hex", pero si dispusiéramos de un programador ISP y en algún momento tuviéramos que "reponer" un bootloader dañado (o bien grabar un bootloader a algún microcontrolador que no tuviera ninguno), Arduino nos ofrece estos ficheros para cargarlos en la memoria Flash de nuestro microcontrolador siempre que queramos.

El formato Intel Hex Format es el utilizado por todos los chips AVR para almacenar contenido en sus memorias Flash, por lo que hay que aclarar que no solamente los bootloaders son alojados internamente de esta forma en la memoria Flash, sino que todos nuestros propios programas que escribamos en el entorno de desarrollo también serán alojados allí en formato ".hex" (aunque de estos detalles no nos debemos preocupar por ahora).

Evidentemente, los bootloaders Arduino también son software libre, por lo que al igual que ocurre con el entorno de programación Arduino, siempre tendremos disponible su código fuente (escrito en lenguaje C) para poder conocer cómo funciona internamente e incluso para poderlo modificar, si así se estima oportuno.

Los gestores de arranque de las placas Due y Zero (ARM)

El microcontrolador de la placa Arduino Due (el SAM3X), a diferencia de los chips AVR, es comercializado por Atmel directamente con un bootloader (llamado SAM-BA) ya precargado. Otra diferencia es que SAM-BA no está grabado dentro de la memoria Flash del microcontrolador sino que está alojado dentro de una memoria especial (una "ROM" –de "Read-Only Memory"–) la cual, como su nombre indica, no permite modificaciones de su contenido: es una memoria de "solo lectura". Esto significa que, a diferencia de lo que ocurría con los microcontroladores AVR, no podremos sustituir el bootloader de la placa Arduino Due por ningún otro.

Por lo demás, el comportamiento "aparente" del bootloader de la placa Due para un usuario del entorno Arduino es (si se usa el "Programming Port") similar al ya conocido, aunque con algunas diferencias: tras reiniciar la placa, SAM-BA solamente se ejecutará si la memoria Flash está vacía; si no, directamente pasará a ponerse en marcha el programa que estuviera grabado en ella; de esta manera, el proceso de arranque de la placa Due es mucho más rápido porque en la mayoría de ocasiones se evita el tiempo de espera provocado por la ejecución del bootloader.

NOTA: Al querer cargar un nuevo programa en la memoria Flash, el entorno Arduino ya procura primero borrar el contenido completo de dicha memoria para así forzar la ejecución de SAM-BA. Por su parte, la placa Due dispone de un botón etiquetado como ERASE para precisamente realizar ese borrado de forma manual.

EL MUNDO GENUINO-ARDUINO

Por su parte, el microcontrolador SAMD21 (en el cual se basa la placa Arduino Zero), a diferencia del SAM3X (y al igual que los de tipo AVR), es comercializado directamente por Atmel sin ningún bootloader preinstalado. No obstante, los microcontroladores incorporados en las placas Zero distribuidas por Arduino sí que vienen (tal como ocurre con las placas de tipo AVR) con un bootloader pregrabado en su memoria Flash, el cual está desarrollado por el propio equipo de Arduino (encontrándose disponible en <https://github.com/arduino/ArduinoCore-samd/tree/master/bootloaders/zero>) a partir del bootloader oficial que ofrece Atmel como referencia, el cual resulta ser otra vez el SAM-BA (pero esta vez descargable de <http://www.atmel.com/devices/ATSAMD21E18A.aspx?tab=documents>). En el caso de querer sobreescribir el bootloader que viene en el micro de la placa Zero (o de querer cargar directamente nuestro programa sin hacer uso de ningún bootloader) deberemos disponer, al igual que ocurría con las placas de tipo AVR, de un aparato hardware específico para ello, como es por ejemplo el llamado "Atmel ICE" (<http://www.atmel.com/tools/atatmel-ice.aspx>).

Otros gestores de arranque más exóticos

Destacaremos, como curiosidad, la existencia de "Ariadne" (<https://github.com/codebendercc/Ariadne-Bootloader>), un bootloader que funciona a través del cable de red TCP/IP en vez de por USB (se necesita, pues, un shield Arduino Ethernet) y compatible con las placas UNO y Mega. Más en concreto, este bootloader implementa un servidor de tipo TFTP que permite la subida del nuevo programa a la placa haciendo uso de un cliente TFTP funcionando en nuestro computador (todos los sistemas operativos proporcionan este tipo de cliente en su instalación por defecto, generalmente en forma de comando de consola). El servidor TFTP de "Ariadne" ya viene con una determinada configuración de red (dirección IP, puerto de escucha, etc.) preestablecida por defecto, la cual se ha de conocer previamente para poder contactar con él (está detallada en su página web); es posible, de todas formas, cambiarla, si así lo deseamos, mediante un código de ejemplo proporcionado por los desarrolladores oficiales. "Ariadne" incluso podría funcionar a través de Internet si los "routers" implicados estuvieran configurados convenientemente. Otro gestor de arranque similar a "Ariadne", desarrollado por Freetronics, está disponible en <https://github.com/freetronics/arduino-tftpboot>

En el caso de no disponer de ningún shield o módulo que aporte la capacidad a nuestra tarjeta Arduino (como pudiera ser el modelo UNO) de conectar a una red TCP/IP, es interesante saber que aún podríamos programarla a una distancia de hasta 100 metros de nuestro computador utilizando para ello igualmente un cable Ethernet y, he aquí el truco, un conversor USB->Ethernet (a colocar en el lado del computador) y Ethernet->USB (a colocar en el lado de la placa), como es el distribuido por Adafruit como producto **nº 2676**.

Otro bootloader que no tiene que ver con los anteriores pero es muy "curioso" y, dependiendo de las circunstancias, hasta bastante práctico, es uno llamado "AVR_boot" (https://github.com/zevero/avr_boot), el cual permite, una vez instalado en el microcontrolador deseado, que un programa previamente compilado (llamado obligatoriamente FIRMWARE.BIN) y que esté ubicado en una tarjeta SD sea automáticamente cargado en la memoria SRAM del microcontrolador al iniciarse este. Es decir, este bootloader ignora la memoria Flash interna del microcontrolador utilizando en su sustitución la memoria ofrecida por la tarjeta SD (que también es de tipo Flash pero es mucho mayor y, además, extraíble). Este gestor de arranque es compatible con todas las placas basadas en el chip ATmega328, el chip ATmega32U4 y el chip ATmega2560.

En cualquier caso, para grabar estos gestores de arranque alternativos (u cualquier otro) deberemos seguir las estupendas instrucciones indicadas tanto en sus respectivas páginas web como, más breviadamente, en la "Breve nota sobre cómo realizar una programación ISP con el entorno Arduino" del apartado sobre el conector ICSP que aparece más adelante en este mismo capítulo.

CARACTERÍSTICAS DE LA PLACA ARDUINO UNO

La placa Arduino UNO, aparte del microcontrolador que incorpora, tiene otras características interesantes a repasar:

La alimentación eléctrica

El voltaje de funcionamiento de la placa Arduino (incluyendo el microcontrolador y el resto de componentes) es de 5V. Podemos obtener esta alimentación eléctrica de varias maneras:

Conectando la placa Arduino a una fuente externa, tal como un adaptador AC/DC o una pila. Para el primer caso, la placa dispone de un zócalo donde se puede enchufar una clavija de 5,5/2,1 milímetros de tipo "jack". Para el segundo, los cables salientes de los bornes de la pila se pueden conectar a los pinos-hembra marcados como "Vin" y "Gnd" (positivo y negativo respectivamente) en la zona de la placa marcada con la etiqueta "POWER". En ambos casos, la placa está preparada para recibir una alimentación de 6 a 20 voltios (aunque, realmente, el rango recomendado de voltaje de entrada –teniendo en cuenta el deseo de obtener una cierta estabilidad y seguridad eléctricas en nuestros circuitos– es menor: de 7 a 12 voltios) porque sea cual sea el valor del voltaje de entrada ofrecido por la fuente externa en ambos

EL MUNDO GENUINO-ARDUINO

casos siempre es rebajado a los 5V de trabajo mediante un circuito regulador de tensión que ya viene incorporado dentro de la placa. El máximo consumo eléctrico que puede realizar de forma segura la placa UNO alimentada de esta manera es de 1A.

Conectando la placa Arduino a nuestro computador mediante un cable USB.

Para ello, la placa dispone de un conector USB hembra de tipo B. La alimentación recibida de esta manera está regulada permanentemente a los 5V de trabajo y es capaz de ofrecer un máximo de hasta 500 mA de corriente (por lo tanto, la potencia consumida por la placa puede alcanzar en ese caso unos 2,5W). Si en algún momento por el conector USB pasara más intensidad de la deseable (porque la placa Arduino la demandara en ese momento determinado), dicha placa está protegida mediante un polifusible reseteable que automáticamente rompe la conexión hasta que las condiciones eléctricas vuelven a la normalidad. Una consecuencia de esta protección contra posibles picos de corriente es que la intensidad de corriente recibida a través de USB puede no ser suficiente para proyectos que contengan componentes tales como motores, solenoides o matrices de LEDs, los cuales consumen mucha potencia.

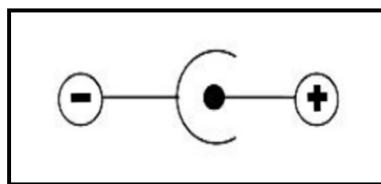
Sea cual sea la manera elegida para alimentar la placa, esta es lo suficientemente "inteligente" como para seleccionar automáticamente en cada momento la fuente eléctrica que esté disponible y utilizar una u otra sin que tengamos que hacer nada especial al respecto. En caso de tener la placa conectada tanto a una fuente externa (adaptador AC/DC o pila) como a nuestro computador vía USB, se alimentará de la fuente externa.

Si utilizamos una pila como alimentación externa, una ideal sería la de 9V (está dentro del rango recomendado de 7 a 12 voltios), y si se utiliza un adaptador AC/DC, se recomienda el uso de uno con las siguientes características:

El voltaje de salida ofrecido por el adaptador ha de ser de 7 a 12V DC. En realidad, el circuito regulador que lleva incorporado la placa Arduino es capaz de manejar voltajes de salida (de entrada para la placa) de hasta 20V, así que en teoría se podrían utilizar adaptadores AC/DC que generen una salida de 20V DC. No obstante, esta no es una buena idea porque se pierde la mayoría del voltaje en forma de calor (lo cual es terriblemente ineficiente) y además puede provocar el sobrecalentamiento del regulador, y como consecuencia dañar la placa.

La intensidad de corriente mínima ofrecida por el adaptador ha de ser de 250mA. Si conectamos a nuestra placa Arduino muchos componentes o unos pocos pero consumidores de mucha energía (como por ejemplo una matriz de LEDs, una tarjeta SD o un motor) el adaptador debería suministrar al menos 500mA o incluso 1A. De esta manera nos aseguraremos de que tenemos suficiente corriente para que cada componente pueda funcionar de forma fiable. Por otro lado, aunque no hay límite para la intensidad máxima que pueda llegar a ofrecer el adaptador, más allá de 1A el regulador de la placa Arduino puede calentarla más de lo debido y volverse inestable; si alguna parte de nuestro circuito necesitara realizar un consumo de corriente mayor que 1A, deberemos emplear entonces una fuente de alimentación separada para esa parte en concreto (independiente de la placa Arduino); veremos cómo en el capítulo 6.

El adaptador ha de ser de polaridad "con el positivo en el centro". Esto quiere decir, como ya sabemos del capítulo anterior, que la parte externa del cilindro metálico que forma la clavija de 5,5/2,1mm del adaptador ha de ser el borne negativo y el hueco interior del cilindro, el borne positivo. Si un adaptador cumple esta condición, lo podremos saber porque tendrá pintado en alguna parte el siguiente símbolo:



Por otro lado, dentro de la zona etiquetada como "POWER" en la placa Arduino existe una serie de pines-hembra relacionados con la alimentación eléctrica, como son:

"GND": pines-hembra conectados a tierra. Es muy importante que todos los componentes de nuestros circuitos comparten una tierra común como referencia y estos pines-hembra realizan esta función.

"Vin": este pin-hembra se puede utilizar para dos cosas diferentes: si la placa está conectada mediante la clavija de 2,1mm a alguna fuente externa que aporte un voltaje dentro de los márgenes de seguridad, podemos conectar a este pin-hembra cualquier componente electrónico para alimentarlo directamente con el nivel de voltaje que esté aportando la fuente en ese

EL MUNDO GENUINO-ARDUINO

momento (¡sin regular por la placa!). Si la placa está alimentada mediante USB, entonces ese pin-hembra aportará 5V regulados. En cualquier caso, la intensidad de corriente máxima que puede aportar es de 400 mA (si está alimentada vía USB) o 900 mA (si está alimentada vía adaptador AC/DC); esto hay que tenerlo en cuenta cuando conectemos dispositivos que consuman mucha corriente, como por ejemplo motores. También podemos usar el pin-hembra "Vin" para otra cosa (ya mencionada en párrafos anteriores): para alimentar la propia placa directamente desde alguna fuente de alimentación externa sin utilizar ni la clavija ni el cable USB. Esto se hace conectándole el borne positivo de la fuente (por ejemplo, una pila de 9V) y conectando el borne negativo al pin de tierra. Si se usa este montaje, el regulador de tensión que incorpora la placa reducirá el voltaje recibido de la pila al voltaje de trabajo de la placa (los 5V); ; en este caso, la intensidad máxima que la placa podrá consumir no deberá exceder de 1A.

"5V": este pin-hembra se puede utilizar para dos cosas diferentes: tanto si la placa está alimentada mediante el cable USB como si está alimentada por una fuente externa que aporte un voltaje dentro de los márgenes de seguridad, podemos conectar a este pin-hembra cualquier componente para que pueda recibir 5V regulados. En cualquier caso, la intensidad de corriente máxima que puede aportar es de 400 mA (si está alimentada vía USB) o 900 mA (si está alimentada vía adaptador AC/DC). Pero también podemos usar este pin-hembra para otra cosa: para alimentar la propia placa desde una fuente de alimentación externa previamente regulada a 5V sin utilizar el cable USB ni la clavija de 2,1mm; en este caso, la intensidad máxima que la placa podrá consumir no deberá exceder de 1A.

"3,3V": este pin-hembra ofrece un voltaje de 3,3 voltios. Este voltaje se obtiene a partir del recibido indistintamente a través del cable USB o de la clavija de 2,1mm, y está regulado (con un margen de error del 1%) por un circuito específico incorporado en la placa: el LP2985. En este caso particular, la corriente máxima que puede aportar es de 50 mA. Al igual que con los pines anteriores, podemos usar este pin para alimentar componentes de nuestros circuitos que requieran dicho voltaje (los más delicados), pero en cambio, no podremos conectar ninguna fuente externa aquí porque el voltaje es demasiado limitado para poder alimentar a la placa.

El chip ATmega16U2

La conexión USB de la placa Arduino, además de servir como alimentación eléctrica, sobre todo es un medio para poder transmitir datos entre nuestro computador y la placa, y viceversa. Este tráfico de información que se realiza entre ambos aparatos se logra gracias al uso del protocolo USB, un protocolo de tipo serie que tanto nuestro computador como la placa Arduino son capaces de entender y manejar. No obstante, el protocolo USB internamente es demasiado complejo para que el microcontrolador ATmega328P pueda comprenderlo por sí mismo sin ayuda, ya que él tan solo puede comunicarse con el exterior mediante protocolos mucho más sencillos técnicamente (como son el I²C o el SPI y pocos más). Por tanto, es necesario que la placa disponga de un elemento "traductor" que permita al ATmega328P (en realidad, el receptor/transmisor serie de tipo TTL-UART que este lleva incorporado y que es quien se encarga realmente de gestionar este tipo de comunicaciones) poder reconocer e interpretar la información transferida vía USB. Esta tarea de "traducir" el protocolo USB a un protocolo serie más sencillo (y viceversa) es, precisamente, la tarea del chip ATmega16U2 que viene incorporado en la placa Arduino UNO.

Breve nota sobre la tecnología TTL y sus niveles HIGH/LOW aceptados

Por "TTL" ("Transistor-to-Transistor Logic") se entiende un tipo genérico de circuito electrónico donde sus elementos de entrada y salida son transistores bipolares. Concretamente el chip UART que viene incorporado dentro del microcontrolador ATmega328P (y que, repetimos, es el encargado real de establecer la comunicación de tipo serie entre este y el exterior) está construido con la tecnología TTL.

Que el receptor/transmisor UART del ATmega328P sea de tipo TTL implica que los valores HIGH (bits "1") recibidos o enviados los representará con un pulso de 5V (o 3,3V, según el voltaje de trabajo del circuito) y los valores LOW (bits "0") los representará con un pulso de 0V. Esto es importante porque otros tipos de receptores/transmisores también serie (como por ejemplo los de tipo RS-232, que no estudiaremos) utilizan otros niveles de voltaje para representar los valores HIGH y LOW (en el caso de los chips RS-232 estaríamos hablando, concretamente, de niveles entre -15V y -3V para el valor HIGH y entre 3V y 15V para el valor LOW, niveles más adecuados para largas distancias) y, por tanto, no compatibles con los niveles TTL (a no ser que utilicemos en medio algún circuito conversor de niveles como, por ejemplo, el conversor TTL<->RS232 "MAX232" del fabricante Maxim, o equivalentes).

En realidad, la explicación dada en el párrafo anterior sobre los niveles HIGH y LOW en circuitos TTL es una gran simplificación porque un circuito que use la tecnología TTL reconocerá como nivel HIGH no solamente el valor exacto de 5V (o 3,3V,

según) sino también un cierto rango de valores menores pero cercanos a este; igualmente, reconocerá como nivel LOW un cierto rango de valores superiores pero cercanos a 0V. La amplitud concreta de estos rangos dependerá de cada circuito en particular, teniendo en cuenta, además, que para un mismo circuito, tanto el rango HIGH como el rango LOW suelen ser diferentes según se esté tratando con señales recibidas (voltaje de entrada) o señales transmitidas (voltaje de salida). Por ejemplo, la implementación TTL en el chip UART del ATmega328P establece los siguientes valores concretos:

- Valor teórico de un nivel HIGH para todas las señales (V_{CC}): 5V.
- Valor mínimo de un nivel HIGH para una señal de salida (V_{OH}): 4,2V (este valor es mayor que el que otro dispositivo funcionando a 3,3V podría soportar como entrada, por lo que en este caso sería necesario colocar al menos un divisor de tensión –o un convertidor de nivel, tal como veremos– entre ese dispositivo y la placa Arduino).
- Valor mínimo de un nivel HIGH para una señal de entrada (V_{IH}): 3V (esto significa que, en teoría, una placa Arduino puede recibir directamente una señal HIGH proveniente de otros dispositivos funcionando a 3,3V sin necesidad de hacer ninguna conversión).
- Valor máximo de un nivel LOW para una señal de entrada (V_{IL}): 1,5V (esto significa que una placa Arduino puede recibir directamente una señal LOW proveniente de otros dispositivos funcionando a 3,3V sin necesidad de hacer ninguna conversión).
- Valor máximo de un nivel LOW para una señal de salida (V_{OL}): 0,9V (este valor es mayor que el que otro dispositivo funcionando a 3,3V generalmente puede reconocer como nivel LOW de entrada, por lo que en este caso sería recomendable colocar al menos un divisor de tensión –o un convertidor de nivel, tal como veremos– entre ese dispositivo y la placa Arduino).
- Valor teórico de un nivel LOW para todas las señales (GND): 0V.

Observar que en la lista anterior existe un rango de voltaje entre 1,5V y 3V no definido ni como HIGH ni como LOW: si se recibe –o envía– una señal dentro de ese rango inválido, el comportamiento de nuestro circuito será totalmente arbitrario.

En realidad, el ATmega16U2 (tal como se puede comprobar en la página oficial del fabricante, <http://www.atmel.com/devices/atmega16u2.aspx>) es todo un microcontrolador en sí mismo (con su propia memoria –tiene, por ejemplo, 16 Kilobytes de memoria Flash para su uso interno, de ahí su nombre–, con su propia CPU, etc.). No obstante, el ATmega16U2 concreto que viene incluido en la placa Arduino UNO viene por defecto con el firmware preprogramado para realizar exclusivamente una función: la de "intérprete" USB<->Serie al ATmega328P, y nada más.

Reprogramación del chip ATmega16U2

Es perfectamente posible desprogramar el chip ATmega16U2 para que, en vez de tan solo realizar la "traducción" entre los protocolos USB<->Serie (y viceversa), pueda realizar muchas otras cosas y convertirse así (y por extensión, convertir a la propia placa Arduino) en virtualmente cualquier tipo de dispositivo USB listo para ser conectado a un computador (como un teclado, un ratón, un joystick, un dispositivo MIDI...).

Desarrollar un nuevo firmware suele ser en general un procedimiento complejo y el caso del chip ATmega16U2 no es una excepción: además de conocimientos avanzados de programación en lenguaje C, se requiere dominar el uso de la librería **LUFA** (www.lufa-lib.org), fundamento sobre el que está basado el código fuente del firmware oficial del microcontrolador ATmega16U2 incorporado en las placas Arduino UNO (y sobre el que debería estar basado cualquier firmware que desarrollemos nosotros mismos para ese chip). Es por esto que este asunto no será abordado en este libro. No obstante, lo que sí es relativamente sencillo es grabar en el chip ATmega16U2 de la placa Arduino UNO un nuevo firmware ya desarrollado previamente, así como volver a grabar el firmware oficial si quisieramos retornar al "statu quo" (o si, por alguna razón, dicho firmware oficial se hubiera corrompido y, por tanto, la comunicación vía USB con la placa fuera imposible).

Para realizar la grabación de un nuevo firmware en el chip ATmega16U2 de una placa Arduino UNO, lo primero que debemos tener a mano es dicho firmware, el cual debe venir siempre en formato ".hex". En concreto, el firmware que viene pregrabado en las placas Arduino UNO oficiales es el fichero llamado "Arduino-COMBINED-dfu-usbserial-atmega16u2-Uno-Rev3.hex" disponible al entrar en la carpeta que contiene el entorno de desarrollo oficial de Arduino y dirigiéndose desde allí a la subcarpeta "hardware/arduino/avr/firmwares/atmegaxxu2" (donde también se halla su correspondiente código fuente por si se desea estudiar o modificar).

NOTA: Existen otros firmwares interesantes (como los que convierten la placa Arduino UNO en un periférico que se puede comportar como un teclado, un ratón o un "joystick", entre otros) que se pueden encontrar –junto con sus correspondientes códigos Arduino de ejemplo– en varias páginas de terceros, como <https://github.com/harlequin-tech/arduino-usb> o <http://hunt.net.nz/users/darran>, y también en la web del propio proyecto LUFA. A destacar, en este sentido especialmente el firmware Hoodloader2, por ser el más flexible y versátil de todos (<https://github.com/NicoHood/HoodLoder2>).

La grabación de un nuevo firmware en el chip ATmega16U2 de una placa Arduino UNO se puede realizar de dos maneras: o bien mediante el protocolo ISP (que estudiaremos en un apartado posterior y que requiere de hardware adicional), o

EL MUNDO GENUINO-ARDUINO

bien mediante el protocolo DFU (de "Device Firmware Update"), el cual funciona a través de una conexión USB estándar. Eso sí, el protocolo DFU necesita, para funcionar, que dentro del "bootloader block" de la memoria Flash del chip ATmega16U2 se encuentre un gestor de arranque compatible. Hay que tener en cuenta, por tanto, no sobrescribir el bootloader DFU presente en el chip cuando se grabe un nuevo firmware: la manera de evitar esta sobrescritura dependerá de cada caso particular, así que es importante leer las instrucciones concretas del programa grabador empleado al reescribir el firmware, así como también conocer las características de ese firmware en cuestión.

Los pasos concretos para realizar la grabación de un nuevo firmware en el chip ATmega16U2 de una placa Arduino UNO mediante el protocolo DFU son:

- 1 En el computador al que conectaremos (mediante el mismo cable USB de siempre) la placa Arduino a reprogramar, tenemos que haber instalado previamente el software grabador necesario para cargar el firmware. Este firmware consiste simplemente en un fichero ".hex" que deberemos tener ya guardado en ese computador. Si estamos ejecutando un sistema Windows, el programa grabador recomendado es el oficial de Atmel, llamado FLIP (<http://www.atmel.com/tools/flip.aspx>) pero si estamos ejecutando Linux –u OS X–, el programa recomendado es el llamado DFU-programmer (<http://dfu-programmer.github.io>, disponible en los repositorios oficiales de las distribuciones más importantes).
- 2 Hay que poner la placa Arduino en "modo DFU" para que sea capaz de aceptar la entrada de un nuevo firmware. Para ello, se deben conectar brevemente (un segundo bastará) dos pines concretos de la placa Arduino UNO entre sí mediante un material conductor cualquiera (así que un "jumper" o incluso un simple cable ya servirá). Estos dos pines son, concretamente, los dos pinchos metálicos (de un grupo de seis) que están más cercanos al conector USB de la placa (los cuales se corresponden, técnicamente, con los conectores RST y GND de tipo ICSP, tal como veremos en próximos párrafos). En este punto, aún no se ha cambiado nada, por lo que se podría abortar el proceso (y volver al "modo normal" de la placa) simplemente desconectándola de nuestro computador.

Breve nota sobre los "pogo pins"

Dependiendo de si hemos adquirido una placa Arduino UNO o bien una placa Genuino UNO, nos podemos encontrar con una diferencia sutil entre ambas pero significativa: que los pinchos mencionados en el párrafo anterior no estén y

aparezcan en su lugar sendos agujeros donde deberíamos soldar esos pinchos, adquiridos aparte (por ejemplo en Sparkfun como producto nº 12807 y en Adafruit como producto nº 2105). Afortunadamente, existe una opción alternativa mucho más rápida y cómoda que nos permitirá utilizar esos pinchos sin tener que soldarlos: emplear "pogo pins". Un "pogo pin" no es más que un pincho con un determinado tipo de cabeza (hay varios tipos: puntiaguda, estrellada, cilíndrica hueca, cilíndrica serrada, etc.) que, en cualquier caso, está diseñado para mantenerse conectado eléctricamente a algún agujero conductor de una placa PCB de forma temporal. Es decir, en vez de soldar permanentemente seis pinchos a los seis agujeros de nuestra placa Arduino UNO, podríamos emplear seis "pogo pins" para conectarlos a dichos seis agujeros solamente mientras fuera necesario (en este caso, solamente para activar el "modo DFU"). De esta manera, se obtiene la misma funcionalidad (temporalmente) pero sin necesidad de soldar. Sparkfun ofrece su producto nº 11591 especialmente diseñado para esta necesidad: se trata de un adaptador de seis "pogo pins" ideales para acoplar a los seis agujeros de nuestra placa Arduino UNO (u otras que carecen de pinchos presoldados, como es el modelo Lilypad). Otro producto similar es el "**Pogo Pin ISCP SPI Programmer Adapter**" de Tindie/FemToCow.

- 3 Una vez establecido el "modo DFU" en la placa Arduino UNO, hay que cargar el nuevo firmware. Para ello, deberemos ejecutar en nuestro computador o bien la aplicación FLIP (si usamos Windows), o bien DFU-programmer (si usamos otros sistemas). En el primer caso, se nos mostrará en pantalla una ventana desde donde, tras seleccionar el dispositivo "ATmega16U2" (gracias a la opción "Select..." del menú "Device"), podremos cargarle el firmware deseado (guardado en forma de fichero ".hex" en nuestro computador, recordemos) seleccionando la opción "Load Hex File" del menú "File", eligiendo a continuación la opción "USB" del menú "Settings->Communication" y, finalmente, pulsando en el botón "Run". En el segundo caso, al tratarse de una aplicación de terminal, deberíamos ejecutar los siguientes tres comandos (como administrador) para realizar el proceso completo de carga:

```
dfu-programmer atmega16u2 erase
dfu-programmer atmega16u2 flash nombreArchivoFirmware.hex
dfu-programmer atmega16u2 reset
```

En el caso de querer (re)cargar el firmware oficial de Arduino, debemos saber que el fichero "Arduino-COMBINED-dfu-usbserial-atmega16u2-Uno-Rev3.hex" antes mencionado no solo incluye el firmware USB<->Serie, sino que también incorpora el propio bootloader DFU (de ahí la palabra

EL MUNDO GENUINO-ARDUINO

"COMBINED"), por lo que al comando *dfu-programmer... flash* anterior deberíamos añadirle el parámetro `--suppress-bootloader-mem`, parámetro necesario para que la carga sobrescriba también el "bootloader block".

- 4 Tras la carga del nuevo firmware, para observar el cambio deberemos desconectar la placa Arduino del zócalo USB de nuestro computador y seguidamente volverla a conectar.

Unos pasos bastante similares a los anteriores pueden seguirse para reprogramar el chip ATmega16U2 de la placa Arduino Due; los detalles se describen aquí: <https://www.arduino.cc/en/Hacking/Upgrading16U2Due>.

Breve nota sobre los VID y PID

Cualquier chip que quiera cumplir con el estándar USB necesita tener un identificador de producto ("product id", PID) y un identificador de fabricante ("vendor id", VID) oficiales, propios y únicos para poder ser comercializado. Los códigos VID son vendidos a los diferentes fabricantes de hardware del mundo por la corporación internacional **USB-IF** (<http://www.usb.org>), encargada de la estandarización y especificación del protocolo USB, y los códigos PID son elegidos por cada fabricante por su cuenta. En el caso del chip ATmega16U2, el VID/PID no está grabado directamente en el hardware (hay otros chips en que esto sí es así, como por ejemplo en los chips FTDI –los veremos más adelante–), sino que dependerá del firmware que tenga preprogramado; si hablamos en concreto del chip ATmega16U2 que viene en las placas Arduino oficiales, su firmware contiene el VID nº 2341, que es el correspondiente al fabricante Arduino.

Las entradas y salidas digitales

La placa Arduino UNO dispone de 14 pines-hembra de entradas o salidas (según lo que convenga) digitales, numeradas desde la 0 hasta la 13. Es aquí donde conectaremos nuestros sensores para que la placa pueda recibir los datos del entorno, y también donde conectaremos los actuadores para que la placa pueda enviarles las órdenes pertinentes (además de poder conectar cualquier otro componente que necesite comunicarse con la placa de alguna manera). A veces a estos pines-hembra digitales de "propósito general" se les llama pines GPIO (de "General Purpose Input/Output").

Es importante tener en cuenta que al ser entradas y salidas digitales, los sensores y/o actuadores que podríamos conectar a ellas deberían ser también de tipo digital (o más propiamente, binario). Esto es así porque los sensores digitales

asimilan cada dato recibido a uno de los dos únicos valores posibles diferentes admitidos en una entrada digital (ALTO y BAJO) y los actuadores digitales están diseñados para interpretar solamente dos órdenes diferentes (ALTO y BAJO) que son las correspondientes a los dos únicos valores posibles admitidos en una salida digital. Si conectáramos un sensor o actuador analógico a una entrada o salida digital, estos funcionarían pero en un modo "blanco o negro" en vez de reconocer la "tonalidades de grises" que se les supone.

Todos estos pines-hembra digitales funcionan a 5V, pueden proveer o recibir un máximo de 40 mA y disponen de una resistencia "pull-up" interna –de entre 20K Ω y 50K Ω – que inicialmente está desconectada (salvo que nosotros indiquemos lo contrario mediante programación software).

Hay que tener en cuenta, no obstante, que aunque cada pin individual pueda proporcionar hasta 40mA como máximo (y la placa puede consumir, en general, hasta 1A), en realidad, los pines digitales están agrupados internamente de tal forma que los pines del nº0 al nº 4 tan solo pueden soportar 100mA en su conjunto (ya sea recibiendo o aportando corriente, según si cada pin en cuestión actúa como entrada o salida) y el conjunto del resto de pines (del nº 5 al nº 13) otros 100mA globalmente. Esto quiere decir que si necesitáramos emplear 10 pines de entrada/salida a la vez, solo podrán recibir/ofrecer hasta 20mA cada uno (es decir, 200mA en total). Una solución para esta limitación es, tal como veremos en el capítulo 6, utilizar transistores.

Las entradas analógicas

La placa Arduino dispone de 6 entradas analógicas en forma de pines-hembra (etiquetados como "A0", "A1"... hasta "A5"), los cuales pueden recibir voltajes dentro de un rango de valores continuos de entre 0 y 5V y una intensidad de hasta 40mA como máximo por pin individual (y 100mA como máximo en su conjunto). No obstante, la electrónica de la placa tan solo puede trabajar con valores digitales, por lo que es necesaria una conversión previa del valor analógico recibido a un valor digital lo más aproximado posible. Esta se realiza mediante un circuito conversor analógico-digital (es decir, un ADC) incorporado en la propia placa.

El ADC es de 6 canales (uno por cada entrada analógica) y cada canal dispone de 10 bits (los llamados "bits de resolución") para guardar el valor del voltaje convertido digitalmente. La cantidad de bits de resolución que tiene un determinado canal del ADC es lo que marca en gran medida el grado de precisión conseguida en la conversión de señal analógica a digital, ya que cuantos más bits de resolución tenga,

EL MUNDO GENUINO-ARDUINO

más fiel será la transformación (aunque, en cualquier caso, siempre estaremos perdiendo algo de información porque estaremos pasando siempre de valores continuos –es decir, "sin cortes"– a solo ciertos valores discretos –es decir, con "saltos"). Por ejemplo, en el caso concreto de los canales del ADC incorporado en la placa Arduino UNO, si contamos el número de combinaciones de 0s y 1s que se pueden obtener con 10 posiciones, veremos que hay un máximo de 2^{10} (1024) valores digitales diferentes posibles. Por tanto, la placa Arduino UNO puede distinguir, partiendo de un voltaje analógico de entre 0V y 5V, un voltaje digital que puede valer desde 0 hasta 1023. Si el conversor tuviera, por ejemplo, 20 bits de resolución, la variedad de valores digitales que podría distinguir para el mismo rango analógico sería muchísimo más grande ($2^{20} = 1048576$) y podría afinar, por tanto, la precisión mucho más.

NOTA: Como se puede ver, para contar de forma sencilla la cantidad de combinaciones posibles dado un determinado número de símbolos diferentes –en el caso binario este número siempre será 2 porque solo hay dos símbolos: el 0 y el 1– y un determinado número de posiciones, se puede utilizar la fórmula $n^{\text{osímbolosdiferentes}}{}^{\text{n^posiciones}}$ (relacionada con las llamadas "variaciones con repetición" matemáticas).

Es fácil entender lo explicado en el párrafo anterior si dividimos el rango analógico de entrada (5V-0V=5V) entre el número máximo posible de valores digitales (1024). Obtendremos que cada valor digital corresponde a una "ventana" analógica de aproximadamente $5V/1024 \approx 5mV$. En otras palabras: todos los valores analógicos dentro de cada rango de 5mV (desde 0 a 5V) se "colapsan" sin distinción en un único valor digital (desde 0 a 1023). Así pues, no podremos distinguir valores analógicos distanciados por menos de 5mV. En muchos de nuestros proyectos ya nos es suficiente este grado de precisión, pero en otros puede que no: si el ADC tuviera más bits de resolución, el resultado de la división *rango_analógico_entrada/número_valores_digitales* sería menor (es decir, la "ventana" sería más pequeña), y por tanto la conversión sería más rigurosa. Pero como no es posible aumentar los bits de resolución del ADC de la placa, si quisiéramos más exactitud, en vez de aumentar el denominador de la división anterior, deberíamos optar por otra solución: reducir el numerador. Es decir, disminuir el rango analógico de entrada, o más específicamente, su límite superior –por defecto igual a 5V–, ya que el inferior es siempre 0. Este límite superior en la documentación oficial se suele nombrar como "voltaje de referencia". La manera práctica y concreta de reducir el voltaje de referencia del ADC de la placa Arduino no la podemos explicar todavía porque aún nos faltan los conocimientos necesarios para poder llevar a cabo todo el proceso; será explicado en el apartado correspondiente del capítulo 6.

Otra característica en la transformación de valores analógicos (es decir, continuos) a digitales (es decir, puntuales) que también influye en la mayor o menor fidelidad del resultado numérico obtenido comparándolo con la señal analógica

original es la "frecuencia de muestreo", llamada así porque es la frecuencia a la que el ADC toma muestras de la señal analógica para convertirlas en valores digitales (con la resolución que toque). Es decir, este factor indica el número de veces por segundo que el ADC toma información (usando en cada muestra los bits de resolución que toque) proveniente de la entrada analógica. Es evidente que cuanto mayor sea la frecuencia de muestreo, la captura de la señal será más fidedigna a la señal real porque dejará menos intervalos sin medir. En el modelo UNO el ADC funciona a un ritmo de 125KHz (dato que proviene de dividir el ritmo del reloj general de la placa –16MHz– entre un factor numérico llamado "prescaler" de valor 128: esta división es necesaria para mantener el número de bits de resolución al máximo, ya que un ritmo mayor lo degradaría) pero como cada conversión tarda 13 ciclos de ese reloj, debemos decir entonces que la frecuencia de muestreo de la placa Arduino UNO es de $125\text{KHz}/13 = 9615 \text{ Hz}$ (o lo que es lo mismo, entre medida y medida ha de transcurrir como mínimo un intervalo aproximado de $100\mu\text{s}$).

Dependiendo de la señal analógica que queramos detectar, la frecuencia de muestreo de la placa Arduino UNO (esos nueve mil hercios aproximadamente) puede ser suficiente para obtener una representación digital fiel al "dibujo" de esa señal, o no. Matemáticamente se puede demostrar que para que dicha representación sea óptima, la frecuencia de muestreo ha de ser como mínimo el doble de la frecuencia propia de la señal que queramos convertir (es lo que se llama Teorema de muestreo de Nyquist-Shannon). Es decir: si, por ejemplo, quisieramos medir una señal que oscila a 60Hz, necesitaríamos un ADC que, al menos, lograra tomar 120 muestras por segundo. Debido a esto, hay que ser consciente de que, por ejemplo, el ADC de una placa UNO no es útil para obtener la representación de ondas sonoras (del sonido), ya que estas oscilan a frecuencias del orden de los kilohercios.

Por último, decir que estos pines-hembra de entrada analógica tienen también toda la funcionalidad de los pines de entrada-salida digitales. Es decir, que si en algún momento necesitamos más pines-hembra digitales más allá de los 14 que la placa Arduino ofrece (del 0 al 13), los 6 pines-hembra analógicos pueden ser usados como unos pines-hembra digitales más (numerándose entonces del 14 al 19) sin ninguna distinción. Por otro lado, también hay que saber que todos los pines-hembra de entrada analógica están agrupados internamente de tal forma que (solo) pueden soportar hasta 100mA en su conjunto.

Las salidas analógicas (PWM)

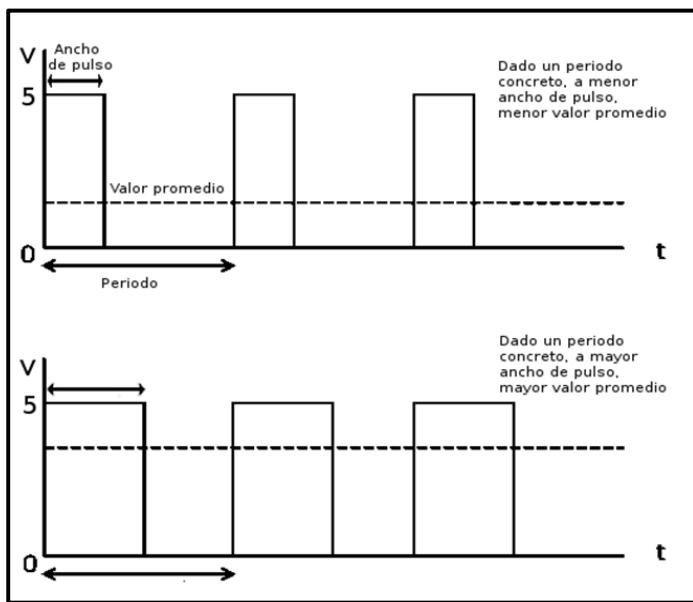
En nuestros proyectos a menudo necesitaremos enviar al entorno señales analógicas, por ejemplo, para variar progresivamente la velocidad de un motor, la

EL MUNDO GENUINO-ARDUINO

frecuencia de un sonido emitido por un zumbador o la intensidad con la que luce un LED; no basta con simples señales digitales: tenemos que generar señales que varíen continuamente, sin saltos. Excepto los modelos Zero y Due (que disponen respectivamente de uno y dos convertidores digital<->analógico –también llamados "DACs"–), las placas Arduino carecen de pines-hembra de salida analógica propiamente dichos (porque su sistema electrónico interno no es capaz de manejar este tipo de señales) así que el "apaño" que hacen es utilizar algunos pines-hembra de salida digitales concretos para "simular" un comportamiento analógico. Los pines-hembra digitales que son capaces de trabajar en este modo no son todos: solo son los marcados con la etiqueta "PWM". En concreto para el modelo Arduino UNO son los pines número: 3, 5, 6, 9, 10 y 11.

Las siglas PWM vienen de "Pulse Width Modulation" (Modulación de Ancho de Pulso). Lo que hace este tipo de señal es emitir, en lugar de una señal continua, una señal cuadrada formada por pulsos de frecuencia constante (aproximadamente de 488Hz –en los pines 3, 9, 10 y 11– o 976Hz, –en los pines 5 y 6–). La gracia está en que al variar la duración de estos pulsos (ojo, no su periodo, que como hemos dicho, es constante), estaremos variando proporcionalmente la tensión promedio resultante. Es decir: cuanto más cortos sean los pulsos, menor será la tensión promedio de salida, y cuanto más largos sean estos, mayor será dicha tensión. El caso extremo lo tendríamos cuando la duración del pulso coincidiera con el periodo de la señal, momento en el cual de hecho no habría "distancia" entre pulso y pulso (sería una señal de un valor constante) y la tensión promedio de salida sería la máxima posible, que son 5V.

Podemos sintetizar lo descrito en el párrafo anterior mediante la siguiente fórmula matemática, la cual nos permite obtener en general el valor de la tensión promedio a partir del valor BAJO de los pulsos (V_B), el valor ALTO de los pulsos (V_A) y el valor del "ciclo de trabajo" (D): $V_{media} = D \cdot V_A + (1 - D) \cdot V_B$. En el caso de la mayoría de placas Arduino, $V_B=0V$ y $V_A=5V$, por lo que la fórmula pasa a ser directamente $V_{media} = D \cdot 5$. El "ciclo de trabajo" se mide en tanto por ciento y se define como la relación entre la duración del pulso –que puede variar– y su periodo –fijo– (o dicho de otra manera, es el porcentaje de tiempo que un pulso está en valor ALTO respecto a su periodo), es decir: $D = (\text{duración_pulso}/\text{periodo_pulso}) \times 100$. Así pues, un ciclo de trabajo del 50% querá decir que la señal es cuadrada perfecta (tenga la frecuencia que tenga) y su valor promedio será de 2,5V. La siguiente figura ilustra lo acabado de explicar:



Hay que insistir en el hecho de que, en general, siempre podremos cambiar la duración del pulso en cualquier momento mientras la señal se esté emitiendo, por lo que la tensión promedio podrá ir variando en consonancia a lo largo del tiempo de forma continua. Esto es precisamente lo que nos permite conseguir las salidas PWM que ofrece la placa Arduino UNO (y el resto de modelos).

Cada pin-hembra PWM de esta placa tiene una resolución de 8 bits. Esto quiere decir que si contamos el número de combinaciones de 0s y 1s que se pueden obtener con 8 posiciones, obtendremos un máximo de 2^8 (256) valores diferentes posibles. Por tanto, podemos tener 256 valores diferentes para indicar la duración deseada de los pulsos de la señal cuadrada (o dicho de otra forma: 256 valores promedio diferentes). Si establecemos (mediante programación software) el valor mínimo (0), estaremos emitiendo unos pulsos extremadamente estrechos y generaremos una señal analógica equivalente a 0V; si establecemos el valor máximo (255), estaremos emitiendo pulsos de máxima duración y generaremos una señal analógica equivalente a 5V. Cualquier valor entremedio emitirá pulsos de duración intermedia y por tanto, tal como ya hemos dicho, generará una señal analógica de un valor entre 0V y 5V.

La diferencia de voltaje analógico existente entre dos valores promedio contiguos (es decir, entre por ejemplo el valor número 123 y el número 124) se puede calcular mediante la división: *rango_voltaje_salida/número_valores_promedio*. En

EL MUNDO GENUINO-ARDUINO

nuestro caso, sería $(5V - 0V)/256 \approx 19,5mV$. Es decir, cada valor promedio está distanciado del anterior y del siguiente por un "saltito" de $19,5mV$. Está claro que cuanto menor fuera esta diferencia, mayor similitud habría entre la señal generada (pretendidamente analógica) y la señal deseada (realmente analógica).

De hecho, al no ser la señal PWM de tipo "realmente" analógico, sus posibles usos son limitados porque solamente se notará su efecto pseudoanalógico si esa señal es enviada a algún dispositivo cuya velocidad de reacción sea menor que la frecuencia PWM utilizada. Es decir: la tensión promedio V_{media} no aparece "mágicamente" de la nada, sino que resulta de cómo reacciona el dispositivo que hayamos conectado a la salida PWM: si ese dispositivo es lo suficientemente rápido como para poder distinguir los pulsos PWM de forma diferenciada, no tendremos una salida analógica sino simplemente una salida digital a una determinada frecuencia. Así pues, los pulsos PWM solamente se interpretarán de forma analógica cuando el dispositivo que los recibe es "lento de reacción" a esos pulsos. Este es el caso, por ejemplo, de los LEDs y de los motores DC. En el primer caso, más que el propio LED la lentitud radica en la capacidad de nuestro ojo de distinguir cambios de iluminación muy rápidos (realizando él entonces el "cálculo" de V_{media} , donde a mayor valor de este, más brillo); en el caso de los motores DC, la inercia del movimiento que realizan hace que durante el tiempo en el que la señal PWM esté en valor BAJO el motor siga moviéndose y durante el tiempo en el que está en valor ALTO se vuelva a alimentar (funcionando, en la práctica, como si recibieran una tensión V_{media} , donde a mayor valor de este, más velocidad de movimiento). Así pues, podemos concluir que los usos más habituales de las señales PWM son dos: la iluminación/atenuación progresiva de LEDs y el control de velocidad variable en motores DC.

Por otro lado, si conectáramos una salida PWM a un altavoz, escucharíamos un tono sonando a la frecuencia de la señal PWM (es decir, a $\sim 500Hz$ o $\sim 1KHz$ según el pin de la placa utilizado, estando ambas del rango de frecuencias audibles, que va aproximadamente de $20Hz$ a $20KHz$) donde, en este caso, el mayor volumen de sonido se obtendría al alcanzar un ciclo de trabajo del 50%. Si quisieramos escuchar otros tonos, deberíamos poder alterar la frecuencia de la señal PWM, algo que, no obstante, el entorno Arduino no permite hacer directamente.

NOTA: En vez de cambiar la frecuencia de una señal PWM, otra opción sería generar "manualmente" (en un pin de salida digital cualquiera) una señal cuadrada con la frecuencia deseada, simplemente jugando con los estados ALTO y BAJO de esa salida digital (tal como se muestra por ejemplo en <https://www.baldengineer.com/software-pwm-with-millis.html>), logrando llegar así a frecuencias de hasta $5,6KHz$. De esta manera, además de poder escuchar otros tonos, en el caso de iluminar/atenuar LEDs el efecto sería más suave.

Finalmente, comentar que, si fuera necesario, una señal PWM se podría transformar en una señal analógica "real" si añadiéramos a la salida que genera dicha señal una determinada circuitería adicional (por ejemplo, un circuito de tipo RC, o bien uno de tipo escalera R-2R, entre otros), pero esto se sale de los objetivos de este libro.

Otros usos de los pines-hembra de la placa

En la placa UNO existen determinados pines-hembra de entrada/salida digital que, además de su función "estándar", tienen otras funciones especializadas. Por ejemplo:

Pin 0 (RX) y pin 1 (TX): permiten que el microcontrolador ATmega328P pueda recibir directamente datos en serie (por el pin RX) o transmitirlos (por el pin TX) sin pasar por la conversión USB-Serie que realiza el chip ATmega16U2. Es decir, estos pines posibilitan la comunicación sin intermediarios de dispositivos externos con el chip TTL-UART del ATmega328P.

En la placa UNO los pines "RX" y "TX" están internamente conectados (mediante resistencias de $1\text{K}\Omega$) al microcontrolador ATmega16U2, por lo que los datos que atraviesen el USB también estarán disponibles en estos pines. Esto significa que es recomendable no usar estos dos pines como GPIO si además vamos a emplear la conexión USB, ya que ambos canales, como acabamos de decir, se comparten. Hay que comentar, de todas formas, que en otros modelos de placa, este hecho de que el canal USB se comunique directamente con los pines "RX" y "TX" no ocurre, por lo que en ese caso ambos elementos actuarían como canales independientes entre sí (esto pasa, por ejemplo, en las placas basadas en el microcontrolador ATmega32U4, el cual es capaz de gestionar él mismo la conexión USB y, por otro lado, controlar también los pines "RX" y "TX").

NOTA: Hay que aclarar que en la placa UNO están incrustados un par de LEDs etiquetados como "RX" y "TX", pero que, a pesar de su nombre, no se encienden cuando se reciben o transmiten datos de los pines 0 y 1, sino solamente cuando se reciben o transmiten datos provenientes de la conexión USB a través del chip ATmega16U2.

Pines 2 y 3: se pueden usar, con la ayuda de programación software, para gestionar interrupciones de tipo hardware. Hablaremos de ellas en el último capítulo del libro.

Pines 10 (SS), 11 (MOSI) , 12 (MISO) y 13 (SCK):se pueden usar para conectar algún dispositivo con el que se quiera llevar a cabo comunicaciones mediante el protocolo SPI. Estudiaremos casos concretos más adelante.

EL MUNDO GENUINO-ARDUINO

Es importante aclarar la función del pin 10 (SS): este pin siempre representa el extremo SS del lado del maestro SPI pero, en realidad, no hay nada que impida utilizar cualquier otro GPIO de la placa para la misma tarea (a diferencia de los otros tres –MISO, MOSI y SCK–, que sí son fijos); de hecho, si hubiera varios dispositivos esclavos, deberíamos recurrir sí o sí a otros pines GPIO para actuar cada uno como un canal SS diferente, conectados a sendos esclavos. No obstante, si no usamos el pin 10 como SS, deberemos tener entonces la precaución de configurar previamente ese pin 10 como salida digital (la manera de hacerlo lo veremos en el capítulo 6), porque si no nuestra placa no podrá actuar como maestro SPI. Y recordemos que el entorno Arduino tan solo permite que las placas actúen como maestro SPI, por lo que no tenemos más remedio que: o bien usar el pin 10 como canal SS para comunicar la placa Arduino con un dispositivo externo (actuando como esclavo), o bien, si se usa otro pin como canal SS, configurar el pin 10 como salida digital (aunque no esté conectada a nada). Esto es debido a que (por razones de funcionamiento interno del ATmega328P) si el pin SS está configurado como entrada (que es, de hecho, como está por defecto), provoca que la placa actúe como esclavo SPI, algo que, desgraciadamente, el entorno Arduino no es capaz de gestionar (peso a que el hardware en realidad sí lo permite).

Por otro lado, hay que saber que la placa Arduino UNO ofrece otro lugar para conectar dispositivos SPI diferente de los pines-hembra mencionados. Ese lugar es el conector ICSP, del cual hablaremos en un próximo apartado. Aclaremos que no se trata de otras líneas MISO, MOSI, etc., diferentes de las existentes en los pines-hembra sino que son las mismas: simplemente aparecen disponibles, por conveniencia, en dos sitios diferentes. La posibilidad de usar el conector ICSP para las comunicaciones SPI es un aspecto muy a tener en cuenta porque la gran mayoría de los modelos de placa Arduino tienen este conector (y siempre podrá ser empleado de esta manera de forma consistente), mientras que el uso de los pines-hembra como conectores SPI varía de modelo a modelo (es decir, no en todas las placas son el nº 10, 11, 12 y 13) e incluso hay hasta modelos de placas que ni siquiera ofrecen ningún pin-hembra como conectores SPI (recayendo, por tanto, esta funcionalidad únicamente en el conector ICSP mencionado).

Pin 13: este pin está conectado directamente a un LED incrustado en la placa (identificado con la etiqueta "L") de forma que si el valor del voltaje recibido por este pin es ALTO (HIGH), el LED se encenderá, y si dicho valor es BAJO (LOW), el LED se apagará. Es una manera sencilla, y rápida de detectar señales de entradas externas sin necesidad de disponer de ningún componente extra.

También existen un par de pines-hembra de entrada analógica que tienen una función extra además de la habitual:

Pines A4 (SDA) y A5 (SCL): se pueden usar para conectar algún dispositivo con el que se quiera llevar a cabo comunicaciones mediante el protocolo I²C/TWI.

El modelo UNO ofrece (por una cuestión de comodidad y ergonomía) una duplicación de estos dos pines-hembra en los dos últimos pines-hembra tras el pin "AREF", los cuales están sin etiquetar porque no hay más espacio físico. Es importante saber esto porque en la mayoría de otros modelos de placas Arduino los únicos pines I²C/TWI disponibles son precisamente estos últimos (eso sí, ya debidamente etiquetados) quedándose los pines A4 y A5 entonces como simples entradas analógicas y punto.

Finalmente, a lo largo de la placa existen diferentes pines-hembra no comentados todavía que no funcionan ni como salidas ni como entradas porque tienen un uso muy específico y concreto:

Pin AREF: ofrece un voltaje de referencia externo para poder aumentar la precisión de las entradas analógicas. Estudiaremos su uso práctico en el capítulo 6.

Pin RESET: si el voltaje de este pin se establece a valor BAJO (LOW), el microcontrolador se reiniciará y se pondrá en marcha el bootloader. Para realizar esta misma función, la placa Arduino ya dispone de un botón, pero este pin ofrece la posibilidad de controlar esta funcionalidad de forma puramente eléctrica

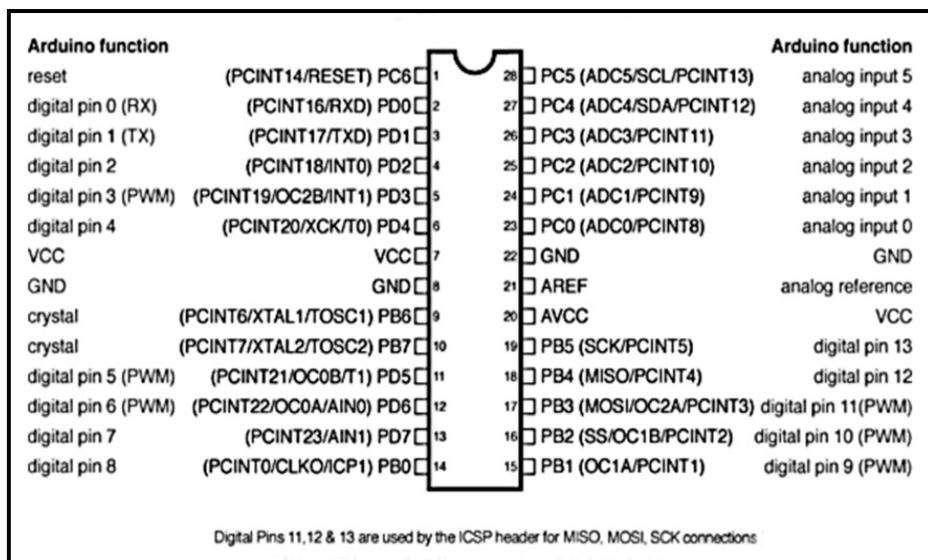
NOTA: Este pin es muy usado por las placas supletorias –es decir, placas que se conectan sobre la placa Arduino para ampliarla y complementarla– para incorporar un botón de reinicio equivalente al "original", por si este queda ocultado bajo ellas).

Pin IOREF: este pin proporciona el voltaje al cual la placa opera (que en el caso del modelo UNO ya sabemos que es 5V pero que en otras placas, como la Zero o la Due, es de 3,3V). Su función es indicar a las placas supletorias conectadas a nuestra placa Arduino ese voltaje de trabajo; de esta manera, una placa supletoria correctamente diseñada podría obtener el valor de voltaje proporcionado por este pin y adaptar entonces su funcionamiento (seleccionando la fuente de alimentación apropiada o activando eventuales convertidores de tensión integrados) para ajustarlo a ese nivel de tensión de trabajo. Es decir, este pin permite a las placas supletorias (que estén preparadas para ello) ser compatibles con diferentes placas Arduino trabajando a diferente tensión.

EL MUNDO GENUINO-ARDUINO

Pin sin utilizar: justo el pin a continuación del IOREF, el cual está sin etiquetar, actualmente no se utiliza para nada, pero se reserva para un posible uso futuro.

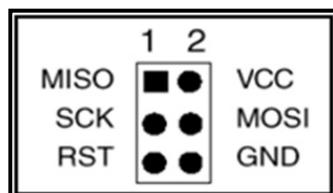
Una vez conocidos todos los pines-hembra de la placa Arduino UNO, es muy interesante observar qué correspondencia existe entre cada uno de ellos y las patillas del microcontrolador ATmega328P. Porque en realidad, la mayoría de estos pines-hembra lo que hacen es simplemente ofrecer de una forma fácil y cómoda una conexión directa a esas patillas, y poco más. Esto se puede ver en la siguiente figura; en ella se muestra cuál es el mapeado de los pines de la placa Arduino respecto a los pines del microcontrolador ATmega328P.



El conector ICSP

SPI

La placa Arduino UNO incorpora un conector ICSP de 6 pines (marcado como tal en su superficie) cuyo diagrama esquemático se muestra en la figura siguiente:



Cada uno de los pines del conector ICSP está empalmado internamente a una patilla concreta del microcontrolador ATmega328P. De la figura anterior podemos comprobar que el conector ICSP implementa en realidad el protocolo estándar SPI. Concretamente, tenemos –además de un pin de alimentación (VCC, el nº2) y otro de tierra (GND, el nº6– el pin de reloj (SCK –"clock"–, el nº3, que marca el ritmo al que se transfieren los datos), el pin de entrada –al microcontrolador, se entiende– de datos (MISO, el nº1), el pin de salida de datos (MOSI, el nº4) y un pin llamado RST (el nº5, conectado al pin RESET del microcontrolador, cuya función no está relacionada realmente con SPI sino con otro protocolo llamado ISP, que estudiaremos en los párrafos siguientes). Fijarse que el canal SS no está presente en el conector ICSP porque, tal como hemos mencionado en párrafos anteriores, para ello se podría emplear cualquier GPIO de la placa (y en el caso del modelo UNO, preferiblemente el pin nº 10, por motivos ya descritos).

NOTA: En realidad, en la placa hay otro conector ICSP (o al menos, los agujeros para acoplarlo), asociado al microcontrolador ATmega16U2. De este otro conector ya se habló en un apartado anterior, que trataba de la reprogramación de este chip mediante el protocolo DFU. Ahora nos olvidaremos de él.

ISP

Las siglas ISP (cuyo significado es "In-Serial Programming") se refieren a un método para programar directamente microcontroladores de tipo AVR que no tienen el bootloader preinstalado. Ya sabemos que la función de un bootloader es permitir cargar nuestros programas al microcontrolador conectando la placa a nuestro computador mediante un simple cable USB estándar, pero si ese microcontrolador no tiene grabado ningún bootloader (o este se ha corrompido), la escritura de su memoria no se puede realizar de esta forma tan sencilla y debemos utilizar otros métodos, como el ISP.

Esta situación nos la podemos encontrar cuando por ejemplo queramos reemplazar el microcontrolador DIP de una placa Arduino UNO por otro que hayamos adquirido por separado sin bootloader incorporado. En este caso podríamos optar por usar el método ISP para grabarle un bootloader (para así volver a poderlo programar vía USB directamente; de hecho, así es como se han grabado precisamente los bootloaders en los microcontroladores de las placas Arduino que vienen de fábrica) o también para grabar directamente siempre nuestros programas sin tener que usar ningún bootloader nunca (con la ventaja de disponer entonces de más espacio libre en la memoria Flash del microcontrolador –el que ocuparía el bootloader si estuviera– y de poder ejecutar nuestros programas inmediatamente después de que la placa reciba alimentación eléctrica sin tener que esperar a la ejecución de un bootloader inexistente). Otra situación en la que ya hemos

EL MUNDO GENUINO-ARDUINO

comentado que nos puede interesar utilizar el método ISP es cuando queramos sobrescribir el bootloader existente por otro, porque el original se haya corrompido o porque queramos sustituirlo por uno diferente más actualizado o capaz.

Para poder programar un microcontrolador adherido a alguna placa Arduino mediante el método ISP se necesita un aparato hardware específico, el "programador ISP". Existen varias formas y modelos, pero hoy en día la versión más extendida de programador ISP es la de un dispositivo cuyo "corazón" interno es un determinado microcontrolador especializado en su función de programador y que consta por un lado de un conector USB que enchufaremos (a través del cable USB pertinente) a nuestro computador y por otro, de un cable acabado en una clavija ICSP lista para ser encajada en el conector ICSP de nuestra placa Arduino. Existen dos estándares para los cables/clavijas/conectores ICSP, uno de 6 pines y otro de 10, pero este último está obsoleto. Normalmente, la clavija ICSP encaja de una sola forma sobre el conector ICSP de la placa, pero para mayor seguridad, el pin correspondiente al conector nº1 (mirar el diagrama anterior) está marcado mediante algún color o muesca diferenciada. El significado de todos los 6 conectores ICSP ya ha sido explicado: básicamente el protocolo ISP es una variante del SPI con tiempos y señales algo diferentes) pero que utiliza casi sus mismas conexiones (MISO, MOSI, SCK, GND, VCC), y una más en sustitución de la SS: la del pin RST; este pin sirve para activar o desactivar la comunicación con el microcontrolador: mientras se reciba un voltaje ALTO no ocurrirá nada, pero cuando se reciba un voltaje BAJO, el ATmega328P detendrá la ejecución del programa que tenga grabado en ese momento y se dispondrá a recibir una reprogramación.

A pesar de que todos tengan una apariencia extensa similar, hay que tener en cuenta que no todos los programadores ISP son compatibles con todos los modelos de microcontroladores AVR. De entre los que lo son con el microcontrolador ATmega328P podemos encontrar los siguientes:

"AVRISP mkII" (<http://www.atmel.com/tools/AVRISPMKII.aspx>): es el programador ISP oficial fabricado por Atmel, basado estrictamente en el protocolo de transferencia STK500 oficial de Atmel. Soporta prácticamente todos los modelos de microcontroladores AVR y los entornos de desarrollo de chips AVR más extendidos (además del entorno Arduino). Existe una versión ya anterior (ya obsoleta) de este programador llamada "AVR ISP".

"USBtinyISP" (<https://learn.adafruit.com/usbtinyisp>): programador ISP (fabricado por Adafruit como producto nº 46) que pretende mejorar el anterior. Por ejemplo, una ventaja que tiene respecto el "AVRISP mkII" es

que, a diferencia de este, permite alimentar con 5V/100mA el microcontrolador a programar (a través de su conexión USB con el computador) sin necesidad de tener, por tanto, que alimentarlo por otro lado. No obstante, no se vende montado sino en forma de kit, por lo que se requiere un mínimo proceso de soldadura de componentes (aunque las instrucciones que se ofrecen son muy claras y detalladas). También se requiere la instalación de un driver si utilizamos el sistema operativo Windows en nuestro computador. DFRobot ofrece una variante de este producto con el código **DFR0116**.

"**USBasp**" (<http://www.fischl.de/usbasp>): programador ISP cuyo esquema hardware y firmware es libre. Requiere la instalación de un driver específico en sistemas operativos Windows. Una versión mejorada de este producto es distribuida por Freetronics con el nombre de "USBASP".

"**Arduino**": si tenemos dos placas Arduino y queremos programar el microcontrolador de una de ellas, podemos conectarlas entre sí para que (comunicándose a través de SPI) el microcontrolador de la primera funcione como programador del de la segunda.

Otros programadores ISP dignos de mención son el "USB AVR Programmer" de Pololu (producto **nº 1300**) y el "USB AVR Programmer" de Seeedstudio (producto con código **TOL132C1B**). También podemos destacar el "**Micro AVR ISP**" de Tindie/NSayer, variante mejorada del producto **nº 9825** de Sparkfun. Si necesitáramos adquirir por separado el cable/clavija ICSP (de 6 pines), podemos recurrir al producto **nº 9215** de Sparkfun o al producto **nº 371** de Adafruit. De todas formas, este cable no es imprescindible porque podemos obtener la misma funcionalidad utilizando cables estándar individuales, cada uno conectado a un conector ICSP diferente de la placa.

Sea como sea, tras conectar correctamente nuestro programador ISP entre nuestro computador y la placa, para programar su microcontrolador es necesario realizar una serie de pasos que involucran el uso de un determinado software ejecutado en nuestro computador, como por ejemplo el propio entorno de Arduino (a través del programa interno *avrdude*). Los pasos concretos a seguir se detallan en la "Breve nota" siguiente, aunque para sacarle el máximo partido recomiendo al lector estudiar primero el capítulo siguiente ya que los pasos descritos aquí suponen que el entorno Arduino ya está instalado y que el lector ya tiene un conocimiento básico de él.

Breve nota sobre cómo realizar una programación ISP con el entorno Arduino

En el caso de disponer de un programador ISP especializado, como el USBtinyISP, el USBasp (o similar), una vez realizadas las conexiones pertinentes (una a la clavija ICSP de la placa Arduino y otra al puerto USB de nuestro computador), deberemos iniciar el software Arduino y seguir los pasos siguientes:

1. Indicar al software Arduino el tipo de programador ISP que va a emplearse mediante la opción *Tools->Programmer->nombre_del_programador*.
2. Seleccionar el modelo de placa/bootloader que va a programarse mediante la opción *Tools->Board*.
3. A partir de aquí, tenemos dos opciones: o bien sobrescribir el bootloader de la placa a programar por uno nuevo, o directamente cargarle un programa propio (sin usar ningún bootloader). Para lo primero, hay que seleccionar la opción *Tools->Burn bootloader* (la cual elegirá el fichero ".hex" correspondiente al bootloader seleccionado en la opción anterior); para lo segundo hay que tener escrito en el entorno Arduino el código a cargar y seguidamente seleccionar la opción *Sketch->Upload using programmer*. En cualquier caso, cuando el proceso haya acabado, el entorno Arduino avisará mediante un mensaje.

En el caso de disponer de una placa Arduino UNO funcional, ya se ha mencionado que también podríamos utilizarla como programador ISP de otra placa. Para ello primero deberemos conectar la placa "programadora" al puerto USB de nuestro computador, iniciar el software Arduino e ir al menú *File->Examples->11.ArduinoISP->ArduinoISP*; al hacer esto se abrirá un código Arduino, el cual, al cargarlo en el microcontrolador (mediante el botón "Upload") transformará la placa Arduino conectada al computador en un programador ISP. Una vez hecho esto, deberemos conectar a la placa "programadora" la placa a programar, mediante la siguiente relación de cables...:

<u>Placa programadora</u>	<u>Placa a programar</u>
Pin "5V"	Pin "5V" o ICSP nº2
Pin "GND"	Pin "GND" o ICSP nº6
MOSI (pin "11" o ICSP nº4)	MOSI (pin "11" o ICSP nº4)
MISO (pin "12" o ICSP nº1)	MISO (pin "12" o ICSP nº1)
SCK (pin "13" o ICSP nº3)	SCK (pin "13" o ICSP nº3)
SS (pin "10")	RST (pin "RESET" o ICSP nº5)

... y tras ello, en el software Arduino deberemos seguir los mismos tres pasos descritos al principio de esta nota.

NOTA: En el paso 1. el nombre del programador a elegir será en este caso "Arduino as ISP";

por otro lado, antes de pasar al paso 2., hay que tener la precaución de asegurarse (mediante la opción *Tools->Port*) de tener seleccionado –por si no lo estuviera ya– el puerto del computador donde está conectado la placa programadora.

En las placas Arduino que no son de tipo AVR (como la Zero o la Due), el protocolo ISP no se utiliza. De hecho, en la placa Due ni tan siquiera se puede sobreescribir el bootloader porque viene en una memoria de solo lectura, separada de la memoria Flash. En cambio, en la placa Zero (que sí admite la sobreescritura de su gestor de arranque) se pueden seguir unos pasos parecidos a los que acabamos de ver: tan solo hay que tener en cuenta que se ha de emplear un dispositivo hardware diferente de los vistos anteriormente (en este sentido, el programador ATMEL-ICE ofrecido por Atmel –<http://www.atmel.com/tools/atatmel-ice.aspx>– es el más popular); este dispositivo se puede conectar por un lado al "Native USB Port" de la placa (y por el otro, como es habitual, a un puerto USB de nuestro computador) y permite utilizar el entorno Arduino para realizar los mismos pasos que los descritos en la "Breve nota" anterior (seleccionando, eso sí, la opción *Tools->Programmer->ATMEL-ICE*).

El reloj

Para marcar el ritmo de ejecución de las instrucciones en el microcontrolador, el ritmo de la lectura y escritura de los datos en su(s) memoria(s), el ritmo de adquisición de datos en los pines de entrada, el ritmo de envío de datos hacia los pines de salidas y en general, para mantener la frecuencia de trabajo del microcontrolador, la placa Arduino posee un pequeño "metrónomo" o reloj, el cual funciona a una frecuencia de 16 millones de hercios (16MHz). Esto quiere decir que (aproximadamente y simplificando mucho) el microcontrolador es capaz de realizar 16 millones de instrucciones en cada segundo. Sería posible que la placa Arduino incorporara un reloj con una frecuencia de trabajo mayor: así disminuiría el tiempo en el que se ejecutan las instrucciones, pero esto también implicaría un incremento en el consumo de energía y de calor generado.

Electrónicamente hablando, existen varios tipos de "relojes". Entre ellos, están los osciladores de cristal y los resonadores cerámicos. Los primeros son circuitos que utilizan un material piezoelectrónico (normalmente cristal de cuarzo, de ahí su nombre) para generar una onda vibratoria de alta frecuencia muy precisa. Los productos de Adafruit nº 2211 –generador de una pulsación de 32768KHz–, nº 2213 –generador de 8MHz–, nº 2214 –de 12MHz– o nº 2215 –de 16MHz– (estos tres últimos van acompañados de un par de condensadores, necesarios para hacer de "by-pass" y de filtro) son ejemplos de este tipo de relojes, así como los productos de Sparkfun nº 539 –generador de una pulsación de 4MHz–, nº 538 –de 8MHz–, nº 535

EL MUNDO GENUINO-ARDUINO

–de 10MHz–, nº 8582 –de 12MHz–, nº 536 –de 16MHz– o nº 534 –de 20MHz–. Los segundos consisten en un material cerámico piezoelectrónico que genera la señal oscilatoria de la frecuencia deseada cuando se le aplica un determinado voltaje. Los productos de Adafruit nº 1872 –generador de una pulsación de 8MHz–, nº 1874 –de 12MHz– o nº 1873 –de 16MHz– son ejemplos de este otro tipo de relojes, así como los productos de Sparkfun nº 542 –generador de una pulsación de 8MHz–, nº 9420 –de 16MHz– o nº 92 –de 20MHz–.

NOTA: Adafruit también distribuye el producto nº 2045, consistente en una plaquita –conectable a la placa Arduino mediante I²C– que incorpora un reloj capaz de generar (a partir de una base de 25MHz) hasta tres pulsaciones diferentes en un rango que va desde los 8KHz hasta los 160MHz; para establecer la frecuencia deseada concreta para cada una de estas tres posibles pulsaciones deberemos utilizar una librería específica, descargable desde https://github.com/adafruit/Adafruit_Si5351_Library.

El reloj que lleva la placa Arduino UNO (o dicho de forma más concreta, que está soldado a determinadas patillas del chip ATmega328) es un resonador cerámico. Este tipo de relojes son ligeramente menos precisos que los osciladores de cristal, pero son más baratos y compactos. En concreto la precisión en la frecuencia aportada por un cristal de cuarzo con compensación de temperatura (los llamados TCXO) es del 0,001% (es decir, con un valor nominal de 16MHz tendríamos valores como mucho de 160Hz por arriba o por abajo de este), pero la precisión aportada por un resonador cerámico típico fabricado con PZT (zirconato titanato de plomo) es del 0,5% (es decir, con un valor nominal de 16MHz tendríamos valores de hasta 80KHz por arriba o por abajo de este). No obstante, esto no es ningún problema porque en cualquiera de nuestros proyectos ya nos será suficiente ese nivel de precisión.

En este sentido, uno podría pensar en utilizar el oscilador de cristal de 16MHz que la placa Arduino UNO incorpora para uso exclusivo del chip ATmega16U2 (dedicado únicamente a mantener la sincronización de la comunicación USB); no obstante, en la práctica esto último generaría demasiado ruido electromagnético y no vale la pena.

Por otro lado, no está de más comentar que el microcontrolador ATmega328P en realidad incluye un reloj interno propio (de tipo RC) dentro de su encapsulado, por lo que en teoría no sería necesario utilizar ningún reloj adicional en la placa Arduino. No obstante, ese reloj interno solo es capaz de marcar un "ritmo" de como mucho 8MHz y además, tiene una precisión muy pobre (un 10%) por lo que solo se suele usar en variantes "caseras" de la placa UNO donde se intentan conectar los mínimos componentes posibles sobre una breadboard/perfboard eliminando, entre otros elementos, la presencia del reloj externo.

NOTA: Para que el microcontrolador siga el ritmo de este reloj interno (en vez del de uno

externo) y para definir su frecuencia concreta (entre muchos otros aspectos) es necesario modificar (mediante un programador ISP y un software especializado como *avrdude* o Atmel Studio) un conjunto de valores avanzados de configuración llamados "fuses" cuya manipulación está fuera del ámbito de este libro. En cualquier caso, hay que saber que alterar la frecuencia del reloj de referencia afecta a las partes del código Arduino sensibles a la cuenta del tiempo, por lo que debe haber un motivo de peso para hacer este cambio.

Los temporizadores ("timers") del microcontrolador

Una de las tareas más elementales que tiene un reloj conectado al microcontrolador de una placa Arduino es controlar los temporizadores de este. Un temporizador (también llamado a veces "contador") es la parte hardware del microcontrolador específicamente encargada de marcar/contar intervalos de tiempo. La utilidad práctica de los temporizadores es tremenda: son fundamentales para generar las señales PWM a una determinada frecuencia, o para controlar los pulsos periódicos que manejan el movimiento de los servomotores, o para definir la frecuencia de una onda sonora (generada mediante la instrucción *tone()*), o para llevar la cuenta del tiempo transcurrido a lo largo de la ejecución del código cargado en la placa (y aprovecharla dentro de él mediante el uso de instrucciones como *delay()*, *millis()*, *micros()*, etc.).

Al ser un aspecto bastante avanzado (y peliagudo), el entorno Arduino no permite la manipulación directa de temporizadores (la cual se realiza alterando directamente los valores de determinados registros). No obstante, a continuación se explicará brevemente su existencia y sentido por considerar que es conveniente que el lector lo conozca para así tener una mejor visión de las capacidades de las placas.

En el caso del microcontrolador ATmega328, por ejemplo, encontramos tres temporizadores (el "Timer0", el "Timer1" y el "Timer2") donde cada uno está especializado en alguna tarea. El "Timer0" es el responsable de llevar la cuenta del tiempo (así que una alteración en su funcionamiento provocará que las instrucciones del lenguaje Arduino basadas en dicha cuenta –como las mencionadas *delay()* o *millis()*, entre otras– no se ejecuten de la forma esperada); el "Timer1" es el responsable de controlar todos los servomotores que estén conectados a la placa; el "Timer2" es el responsable de generar la señal acústica a la frecuencia adecuada a partir de la instrucción *tone()*; además, cada uno de ellos es responsable también de producir una determinada señal PWM (el "Timer0" de 977Hz y los "Timer1" y "Timer2" de 488Hz).

Cada temporizador del ATmega328 está formado por dos canales. Esto significa que cada uno puede estar (y, de hecho, está) conectado a dos pines-hembra

EL MUNDO GENUINO-ARDUINO

de la placa UNO. Concretamente, el "Timer0" está conectado a los GPIO nº 5 y 6, el "Timer1" a los GPIO nº 9 y 10 y el "Timer2" a los GPIO nº 3 y 11. Es por esto que la placa UNO tiene 6 GPIO con capacidad PWM. Desgraciadamente, esta es la causa también de que si se utilizan servomotores no se puedan a la vez generar señales PWM en los pines-hembra nº9 y nº10 o que si se produce audio mediante `tone()`, no se puedan a la vez generar señales PWM en los pines nº3 y nº11.

No todos los temporizadores son iguales: según la cantidad de bits que puedan manejar en los registros del microcontrolador para representar el paso del tiempo, tendrán una resolución más o menos precisa. Concretamente, en el caso del ATmega328, los temporizadores "Timer0" y "Timer2" son de 8 bits (y por tanto, pueden distinguir hasta 256 valores de "pulsaciones" diferentes) y el "Timer1" es de 16 bits (pudiendo distinguir entonces hasta 65536 valores).

En el caso de la placa Arduino Mega (es decir, del microcontrolador ATmega2560) disponemos de tres temporizadores de dos canales (también llamados "Timer0" –de 8 bits–, "Timer1" –de 16 bits– y "Timer2" –de 8 bits– y también asociados, como en la placa UNO, a la cuenta de tiempo, al control de servomotores y a la generación de tonos acústicos, respectivamente) y de tres temporizadores más ("Timer3", "Timer4" y "Timer5", todos de 16 bits) de tres canales; esto implica que podamos tener un total de 15 pines-hembra con capacidad para generar señales PWM. En el caso de las placas que incorporan el microcontrolador ATmega32U4 disponemos, en cambio, de cuatro temporizadores: "Timer0" (también asociado a la cuenta del tiempo), "Timer1", "Timer2" y "Timer3"; en el caso de la placa Due disponemos de 9 temporizadores, todos ellos de tres canales y de 32 bits, etc., etc. Como se puede ver, la casuística es muy variada pero, afortunadamente, el entorno Arduino nos oculta esta complejidad.

El botón de "reset"

La placa Arduino UNO (y, de hecho, la gran mayoría de placas Arduino) dispone de un botón de reinicio ("reset") que permite, una vez pulsado, enviar una señal LOW al pin "RESET" de la placa para parar y volver a arrancar el microcontrolador. Como en el momento del arranque del bootloader siempre se activa, precisamente, la ejecución del bootloader, el botón de reinicio se suele utilizar para permitir la carga de un nuevo programa en la memoria Flash del microcontrolador –eliminando el que estuviera grabado anteriormente– y su posterior puesta en marcha. No obstante, en la gran mayoría de placas Arduino (como por ejemplo es el caso de la UNO) no es necesario prácticamente nunca pulsar "real y físicamente" dicho botón antes de cada carga, ya que estas placas están diseñadas para activar el bootloader (es decir, enviar una señal LOW al pin "RESET")

CAPÍTULO 2: HARDWARE GENUINO

directamente desde el entorno de desarrollo instalado en nuestro computador con tan solo pulsar sobre un determinado ícono de dicho entorno (el ícono "Upload").

Si se desea, se puede deshabilitar esta función de "auto-reset", de tal forma que siempre sea necesario utilizar el pulsador incorporado en la placa para ejecutar el bootloader y, por tanto, cargar un nuevo programa en el microcontrolador. Existen varias maneras para lograrlo, pero la más sencilla es conectar un condensador de 10 microfaradios entre los pines "GND" y "RESET" de la placa. Otra manera más definitiva de quitar el "auto-reset" sería cortar la traza etiquetada como "RESET-EN" en el dorso de la placa (para volver a tener el "auto-reset", cada extremo de esa traza debería ser soldado de nuevo entre sí). En todo caso, una vez deshabilitado el "auto-reset", el procedimiento de carga de nuestros programas sería el siguiente:

1. Mantener apretado el pulsador de reinicio de la placa.
2. Clicar en el botón "Upload" del entorno de desarrollo Arduino.
3. Tan pronto como se ilumine una vez el LED etiquetado como RX en nuestra placa Arduino, rápidamente soltar el pulsador.
4. La carga debería empezar haciendo parpadear a los LEDs RX y TX.

Obtener el diseño esquemático y de referencia

Los curiosos (con un nivel avanzado de electrónica) que deseen conocer hasta el mínimo detalle cómo está construida la placa Arduino UNO y cómo están interconectados los diferentes componentes que la forman, pueden descargarse el diseño esquemático en formato pdf de la siguiente dirección:
http://arduino.cc/en/uploads/Main/Arduino_Uino_Rev3-schematic.pdf

También está disponible el diseño de referencia necesario para construir nosotros mismos una placa de circuito impreso que sea exactamente igual a la de la Arduino oficial (o bien modificada si modificamos estos archivos) y completarla posteriormente añadiendo los componentes necesarios (microcontroladores, pines-hembra, etc.) por separado. Este diseño puede descargarse desde la siguiente dirección: http://arduino.cc/en/uploads/Main/arduino_Uino_Rev3-02-TH.zip. Si descomprimimos este archivo zip, veremos que contiene dos ficheros: el fichero .SCH es un diagrama visual esquemático de la PCB que contiene una serie de símbolos gráficos representando principalmente puertas lógicas y líneas de conexión cuya utilidad es generar fácilmente a partir de él el fichero .BRD, que es el que realmente contiene todos los datos necesarios para la fabricación de la PCB.

El contenido de estos archivos se puede ver y editar mediante un software (no libre) de diseño de PCBs llamado EAGLE –versión 6.0 o posterior–. Se puede descargar de su página oficial <http://www.cadsoftusa.com> una versión de prueba gratuita de 30 días.

¿QUÉ OTRAS PLACAS ARDUINO OFICIALES EXISTEN?

A continuación, haremos un breve resumen de las posibilidades que ofrecen las otras placas Arduino oficiales diferentes del modelo UNO. Todos estas variantes están especializadas en trabajar dentro de circunstancias específicas donde la placa UNO estándar no nos ofrece soluciones a las necesidades que nos puedan surgir. Por ejemplo, puede haber ocasiones en las que necesitemos una placa con más cantidad de GPIOs (en ese caso, sería recomendable emplear el modelo Mega), o con más memoria y mayor velocidad de CPU (aquí destacan los modelos Zero o Due), o con un tamaño más reducido (los modelos Pro Mini, Micro o Nano serían ideales), o con capacidad para conectarse al exterior vía Wi-Fi o Bluetooth (como los modelos Yún o 101), o que permita ser cosida a tela (esto es lo que permiten los modelos de la familia Lilypad o Gemma), etc.

La buena noticia es que, utilicemos el modelo de placa Arduino que utilicemos, cualquiera de ellas es programada mediante el mismo entorno y lenguaje Arduino (con solo pequeñas variaciones dependiendo de las funcionalidades disponibles). Esto es muy importante, porque hace que siempre podamos trabajar de la misma manera, abstrayéndonos (en términos generales) de las especificidades del hardware utilizado. De hecho, aquí radica la grandeza de Arduino: ofrecer una forma homogénea y coherente de realizar proyectos que, por naturaleza, pueden requerir montajes muy diferentes entre sí.

Si se desea obtener información más detallada y completa de la que se proporciona en los párrafos siguientes, lo mejor es consultar el enlace siguiente: <http://arduino.cc/en/Main/Products>, donde se especifican todos los datos técnicos de cada una de estas placas.

Arduino Pro

Esta placa, diseñada en colaboración con Sparkfun, se distribuye en dos "versiones": ambas contienen un microcontrolador Atmega328P (como el del modelo UNO SMD, y por tanto, ofrecen la misma cantidad de memoria SRAM, Flash y EEPROM), pero una funciona con 3,3V y a 8MHz (producto nº 10914 de Sparkfun) y la otra funciona con 5V y a 16MHz (producto nº 10915 de Sparkfun). En cualquier caso, las dos versiones disponen de 14 agujeros, cada uno de los cuales deberá ser soldado directamente a un cable (o, si se desea, a un pin-hembra de plástico) para poderlo utilizar como pin de entrada/salida digital (6 de esos agujeros también pueden funcionar como salida PWM). Además, disponen de 6 agujeros listos para ser empleados (si se suelda el cable o pin-hembra pertinente) como entradas analógicas,

los agujeros necesarios para montar un eventual conector de alimentación de 5,5/2,1mm, un zócalo JST de 2 pines (novedad!) para poder recibir alimentación de una batería LiPo externa (también podrían utilizarse para ello –previa regulación de tensión– sus pines "Vcc" y "Gnd"), un interruptor de corriente, un botón de reinicio, un conector ICSP y, finalmente, unos "pinchos" especiales de los cuales hablaremos en el subapartado siguiente.

Esta placa está pensada para instalarse de forma semi-permanente en objetos o exhibiciones. Por eso no viene con los pines-hembra montados: para permitir el uso de diferentes tipos de configuraciones y orientaciones según las necesidades.

Los pines-hembra (y otros)

A lo largo de este libro nos encontraremos con diferentes placas (y placas supletorias, también llamadas "shields") que no llevan soldado ningún pin-hembra (es decir, tan solo llevan, como el modelo Pro, los agujeros de las conexiones pertinentes). En estos casos, a menudo es conveniente primero soldar una ristra de pines-hembra a dichos agujeros para proceder entonces al montaje y conexionado efectivo de nuestro proyecto. Aunque en este libro no se tratará el tema de la soldadura de componentes, sí deberemos conocer algunos conceptos sobre los distintos tipos de pines-hembra existentes y sus distintas posibilidades.

NOTA: En realidad, el uso de los pines-hembra no es estrictamente necesario porque –por ejemplo– también se podrían soldar directamente los cables a los agujeros, pero esto no es una solución tan flexible ni cómoda.

En este sentido, nos podemos encontrar con diferentes tipos de ristras de pines-hembra. Todas las que listaremos aquí están diseñadas con una distancia de 2mm (0,1 pulgada) entre pines, que es la distancia estándar entre agujeros de una breadboard y también entre los agujeros (de I/O, alimentación, etc.) de las placas Arduino y compatibles. Por otro lado, esas ristras pueden tener un número fijo ya predefinido de pines o bien (si se tratan de las llamadas "break-away" o "break-apart") pueden permitir el fácil desacople de los pines existentes, posibilitando así la obtención de ristras con número de pines a medida, incluso pines individuales.

De entre las ristras mencionadas a continuación puede haber productos (los llamados "stackable") que ofrezcan en su parte superior pines-hembra (es decir, agujeros de plástico donde es posible conectar fácilmente elementos externos) y otros productos que no (los llamados "male"). En ambos casos, no obstante, en su parte inferior siempre sobresalen unos pinchos metálicos que son los que se deberán soldar –usando la técnica THT– a la superficie dorsal de la placa PCB sobre la que se

EL MUNDO GENUINO-ARDUINO

vayan a colocar; estos pinchos son los que se acoplarán, una vez ya soldados, a los agujeros de una breadboard (o los agujeros de una ristra de pines-hembra inferior) para establecer entonces las conexiones pertinentes.

La placa Arduino Pro (siguiendo el mismo esquema que la UNO R3) incorpora una ristra de 6 agujeros, 2 de 8 agujeros y una de 10 agujeros. Así pues, a esta placa (y también, de hecho, a todas aquellas placas supletorias que tengan agujeros en vez de los pines correspondientes ya soldados) les deberemos soldar nosotros mismos ristras con ese mismo número de pines-hembra. En este sentido, Sparkfun distribuye el producto **nº 11417**, consistente precisamente en un pack de 1 ristra "stackable" de 6 pines-hembra (producto **nº 9280**), 2 de 8 (producto **nº 9279**) y 1 de 10 (producto **nº 11376**) y Adafruit distribuye el producto **nº 85**, el cual es similar al pack de Sparkfun pero incorpora además un zócalo 2x3 (ideal para añadir un conector ICSP a nuestra placa supletoria). Freetronics distribuye un producto idéntico al pack de Adafruit con el nombre de "**Stackable Arduino Shield Headers**". Destacaremos también la existencia del producto **nº 743** de Sparkfun, consistente en una ristra "stackable" de tipo "break-away", de manera que podemos crear nuestras propias tiras de pines-hembra con el número deseado de ellos en cada una.

Por otro lado, también conviene tener a mano ristras "break-apart" de tipo "male", como por ejemplo los productos de Adafruit **nº 2671** (similar al **nº 116** de Sparkfun o al **FIT0084** de DFRobot), **nº 400** (similar al **FIT0105** de DFRobot) o **nº 392** (similar al **nº 1540**, también de Adafruit –o al **nº 553** de Sparkfun–, pero estos últimos ofrecen los pines en ángulo recto).

Los adaptadores USB-Serie

Lo que tal vez llame más la atención de la placa Arduino Pro es que no dispone de zócalo USB de ningún tipo. Esto quiere decir que desde nuestro computador no podemos comunicarnos mediante USB con el bootloader de nuestro microcontrolador para poderle grabar en su memoria Flash el programa que deseemos que execute. Para resolver este obstáculo, podríamos utilizar un programador ISP, pero lo más sencillo es adquirir y emplear un adaptador USB<->Serie. Este tipo de adaptadores (que pueden venir en forma de plaquita o bien en forma de cable) se han de enchufar por un lado a los seis "pinchos" que sobresalen de la placa Arduino Pro (cada uno de los cuales está conectado a su vez directamente a una determinada patilla del ATmega328P: el pincho RX a la patilla TX, el pincho TX a la patilla RX, el pincho de alimentación, el de tierra y el de reinicio del micro –marcado como DTR o, a veces, GRN–) y por otro, o bien a un cable USB micro-B/mini-B<->A que deberemos conectar a su vez a nuestro computador (si usamos un adaptador en forma de plaquita), o bien directamente al zócalo USB tipo A de

nuestro computador (si usamos un adaptador en forma de cable). De esta manera, el adaptador USB<->Serie recién añadido nos permitirá tanto alimentar la placa eléctricamente como programarla (incluyendo la capacidad de realizar el auto-reset) de forma similar a como se consigue en el modelo UNO, vía USB.

Ejemplos de adaptadores USB<->Serie en forma de plaquita son, por ejemplo, los productos de Sparkfun nº **9873** (que ofrece una tensión de 3,3V), nº **9716** (que ofrece 5V) o nº **12731** (compatible con ambas tensiones), el producto nº **284** de Adafruit (compatible también con ambas), los productos **DFR0065** o **DFR0164** de DFRobot (también compatibles con ambas), el "**USB Serial Adapter**" de Freetronics, el "**FTDI Adapter**" de Akafugu, el "**Foca**" de IteadStudio, el "**USB-BUB-II**" de ModernDevices, el "**FT232RL USB to UART Breakout Board**" de Gravitech, etc., etc. Todos ellos –excepto el DFR0164 y la plaquita de Freetronics– incorporan, para realizar su función, el chip FT232RL (o una variante de este) del fabricante FTDI (<http://www.ftdichip.com>), chip específicamente diseñado para realizar este tipo de conversión USB<->Serie.

NOTA: Las excepciones incluyen, en cambio, o bien el mismo chip que viene en la placa Arduino UNO –es decir, el ATmega16U2– con el mismo firmware ya cargado preparado para realizar la misma función (ese es el caso del producto de Freetronics), o bien el chip ATmega8U2, un chip muy similar al ATmega16U2 pero con la mitad de memoria Flash –de ahí su nombre– también con el firmware adecuado ya cargado (ese es el caso del producto de DFRobot).

Ejemplos de adaptadores USB<->Serie en forma de cable son, por ejemplo, el producto nº **70** de Adafruit (compatible con 3,3V y 5V), los productos de Sparkfun nº **9717** (compatible con 3,3V) y nº **9718** (compatible con 5V), el producto **FIT0416** de DFRobot (compatible también con 5V) o los propios productos oficiales de FTDI dentro de la gama "USBTTLSerial" (concretamente los modelos "**TTL-232R-5V**" –compatible con 5V– y "**TTL-232R-3V3**" –compatible con 3,3V–).

Arduino Pro Mini

Esta placa, también diseñada en colaboración con Sparkfun, se distribuye en dos "versiones": ambas contienen un microcontrolador Atmega328P (como el del modelo UNO, concretamente de tipo SMD), pero una funciona con 3,3V y a 8MHz (producto nº **11114** de Sparkfun) y la otra funciona con 5V y a 16MHz (producto nº **11113** de Sparkfun).

Al igual que ocurre con la/s placa/s Arduino Pro, ambas versiones de la placa Arduino Pro Mini disponen de 14 agujeros donde se deberá soldar una ristra de pines-hembra (o directamente a un cable) para conseguir funcionar como pines de

EL MUNDO GENUINO-ARDUINO

entrada/salida digital (6 de esos agujeros pueden ser usados como salida PWM), así como también 6 agujeros para entradas analógicas, un botón de reinicio y los agujeros necesarios –que no pines- para conectar un adaptador o cable USB-Serial y así poder programarla (y también alimentarla) directamente vía USB.

Al igual que la/s placa/s Arduino Pro, el modelo Arduino Pro Mini está pensado para instalarse de forma semi-permanente en objetos o exhibiciones. Por eso no viene con los pines montados sino que hay colocar en los agujeros los pines-hembra de plástico "a mano" (o bien soldar cables directamente). De esta manera, se permite el uso de diferentes tipos de configuraciones según las necesidades.

La característica más destacable de la placa Arduino Pro Mini (y su diferencia fundamental respecto a su "pariente mayor", la Arduino Pro) es su reducido tamaño, de tan solo 18 milímetros de anchura por 33 milímetros de longitud. Es por ello que esta placa carece de algunos elementos de la Arduino Pro como son el conector ICSP, los agujeros para montar un eventual conector de alimentación de 2,1mm y el zócalo JST para una batería LiPo externa. En este sentido, la manera de alimentar la placa Arduino Pro Mini solo puede ser o bien a través del conector FTDI vía USB, o bien a través de los agujeros "Vcc" y "Gnd" (en este último caso, hay que tener cuidado porque el voltaje aportado a la placa ha de estar previamente regulado al valor de la tensión de trabajo de la placa –3,3V o 5V según la variante–; de todas formas, si la fuente no aportara una tensión regulada, mientras esta sea menor de 12V se podría emplear el agujero marcado como "RAW" –en vez del "Vcc"– gracias al regulador conectado a él).

Arduino Nano

Esta placa, diseñada en colaboración con Gravitech, contiene el microcontrolador Atmega328P funcionando a 16MHz y 5V (igual, por tanto, que en modelo UNO, concretamente de tipo SMD). De hecho, la placa Arduino Nano sigue ofreciendo el mismo número de salidas y entradas digitales y analógicas que la placa Arduino UNO y la misma funcionalidad que esta. Por tanto, la característica más destacable y diferenciadora de esta placa es su reducido tamaño, de tan solo 18 milímetros de anchura por 45 milímetros de longitud. Estas dimensiones se consiguen eliminando de esta placa el conector de alimentación de 5,5/2,1mm (alimentándose, por tanto, a través del pin "Vin" –fuente no regulada– o "5V" –fuente regulada– y "Gnd") e incorporando un conector USB mini-B en vez del conector USB tipo B y el conversor USB<->Serie FTDI FT232RL en vez del chip ATmega16U2.

Esta placa está especialmente pensada para ser conectada a una breadboard mediante las patillas que sobresalen de su parte posterior, pudiendo formar parte así de un circuito complejo de una manera relativamente fija.

Arduino Mega 2560

Placa basada en el microcontrolador ATmega2560 y cuyas características más destacables son sus 54 pines de entrada/salida digitales (de los cuales 14 pueden ser usados como salidas analógicas PWM), sus 16 entradas analógicas y sus 4 receptores/transmisores serie TTL-UART, además de disponer de una memoria Flash de 256 Kilobytes (de los cuales 8 están reservados para el bootloader), una memoria SRAM de 8KB y una EEPROM de 4KB. Su voltaje de trabajo es igual al del modelo UNO (5V) y el límite máximo de corriente admitida por pin es también igual (40mA).

En general, esta placa nos será útil en proyectos donde el modelo UNO se nos quede corto porque se necesiten más entradas/salidas y/o más memoria (de cualquier tipo: Flash, SRAM o EEPROM). Hay que tener en cuenta, no obstante, que sus dimensiones son mayores (depende del proyecto esto puede ser un factor limitante) y que es posible que algunas placas supletorias no sean compatibles ya que muchas de ellas están diseñadas para el modelo UNO en particular.

En la sección correspondiente a esta placa dentro de la web oficial de Arduino podemos descargarnos los ficheros del diseño esquemático de la placa en PDF y los ficheros del diseño de la PCB en el formato propio del programa EAGLE, así como la documentación oficial del microcontrolador ATmega2560. Allí también aparece un enlace a la página <https://www.arduino.cc/en/Hacking/PinMapping2560>, la cual muestra una imagen ilustrando el mapeado de los pines del microcontrolador con relación a los pines de la placa.

Arduino Micro

Esta placa, diseñada en colaboración con Adafruit, también es, como los modelos Pro Mini y Nano, de un tamaño reducido (concretamente de 18 milímetros de anchura por 48 milímetros de longitud) y, por ello carece de conector 5,5/2,1mm, pero aporta una novedad muy importante: el microcontrolador que lleva incorporado es diferente al de los otros modelos anteriormente vistos porque se trata del chip ATmega32U4 (trabajando a 16MHz). Este micro mantiene la mayoría de funcionalidades que ofrece el Atmega328P (misma tensión de entrada admitida –entre 6V y 20V– y recomendada –entre 7V y 12V–, misma tensión de trabajo –5V–, misma cantidad de memoria Flash y EEPROM –32KBytes y 1Kbyte, respectivamente–, misma capacidad para comunicarse a través de los protocolos SPI, I²C/TWI o Serial,

EL MUNDO GENUINO-ARDUINO

mismos pines "Vin", "5V", "3.3V" y "GND" de alimentación, un número algo mayor de entradas/salidas digitales, entradas analógicas y salidas PWM –20, 12 y 7, respectivamente— aunque con mismos límites máximos de corriente admitida, etc.) pero incorpora además 0,5 kilobytes más de memoria SRAM y, sobre todo, soporta comunicaciones USB directamente (y por tanto, no necesita de ningún chip suplementario como el ATmega16U2 o el FTDI).

Este hecho permite que, además de poder programarla (y alimentarla) mediante la conexión vía USB establecida con nuestro computador (como ocurre con la mayoría del resto de modelos de placas Arduino), podamos utilizar el –único– zócalo USB de tipo micro-B del que dispone la placa para hacerla funcionar (habiendo cargado previamente el programa pertinente) como teclado, ratón o "joystick". Dicho de otra manera, que esta placa solo incluya un microcontrolador tanto para ejecutar los programas como para comunicarse directamente a través de USB con el computador (esta última funcionalidad es lo que se llama tener "USB nativo") permite que pueda simular fácilmente ser (si se programa convenientemente) un dispositivo USB conectado a dicho computador. Esto abre la puerta a que la placa Micro pueda, por ejemplo, interactuar con cualquier aplicación de escritorio, grabar datos sobre un archivo de texto u hoja de cálculo directamente, etc., etc.

NOTA: Técnicamente, cuando la placa Micro se conecta con un cable USB al computador, este detecta dos "puertos" de conexión diferentes: un puerto USB estándar listo para usar dicha placa como un periférico USB más (lo típico sería un teclado o un ratón, tal como hemos dicho) y otro puerto diferente, similar al generado cuando se conecta la Arduino UNO, usable de la forma "tradicional", para la programación y comunicación con la placa a través del entorno de programación Arduino.

Al igual que el modelo Nano, la placa Arduino Micro está especialmente pensada para ser conectada sobre una breadboard (sin ocupar apenas espacio) mediante las patillas que sobresalen de su parte posterior, pudiendo formar parte así de un circuito complejo de una manera relativamente fija. Dispone además de un botón de "reset" y de un conector ICSP. Desgraciadamente, el chip ATmega32U4 solo está disponible en formato SMD, por lo que no permite su conexión directa sobre breadboards independientemente de la placa Micro (u otras plaquitas similares).

Al igual que ocurre con el resto de modelos de placas Arduino, en la sección correspondiente a la placa Micro dentro de la web oficial podemos descargarnos sus ficheros del diseño esquemático en PDF y los ficheros del diseño de la PCB en el formato propio del programa EAGLE, así como la documentación oficial del microcontrolador ATmega32U4. Allí también aparece un enlace a la página <https://www.arduino.cc/en/Hacking/PinMapping32u4>, la cual muestra una imagen ilustrando el mapeado de los pines del microcontrolador con relación a los pines de la placa.

El "auto-reset" del micro ATmega32U4

Debido también a que la placa Micro (y, tal como en seguida veremos, también las placas Yún y LilypadUSB) utiliza un único microcontrolador tanto para la ejecución de los programas como para la comunicación USB con el computador, al reiniciar el microcontrolador (apretando físicamente el botón de "reset" de la placa o bien a través del botón correspondiente del entorno de desarrollo) la conexión USB similar a la UNO (la otra no) se interrumpe y se vuelve a restablecer; esto no ocurre con la placa UNO porque allí la conexión USB la mantiene siempre un chip independiente –el ATmega16U2– que nunca es reiniciado. La consecuencia más importante de esto es que la carga de los programas se demora unos cuantos segundos, ya que el entorno de desarrollo Arduino debe esperar hasta que se detecte otra vez una conexión USB activa: solo entonces podrá comunicarse con el bootloader para ordenarle que efectúe la carga del programa. Si se utiliza el botón de "reset" físico, esta demora implica que deberemos mantener pulsado este botón durante esos segundos hasta que veamos el mensaje "Uploading..." en la barra inferior del entorno de desarrollo: no lo podremos soltar antes.

Afortunadamente, en el arranque de la placa no hay demora para ejecutar el programa grabado porque el "auto-reset" de las placas Micro, Yún y LilypadUSB solo se activa cuando el computador (o el botón físico) le envía una señal serie específica a 1200 bits/s: si la placa Arduino no recibe esa señal, el bootloader no se ejecutará y por tanto, empezará a ponerse en marcha nuestro programa Arduino directamente. NOTA: Conviene indicar aquí también que los puertos USB "nativos" de las placas Zero y Due (hablaremos de ellos en próximos apartados) responden a esa misma señal serie de 1200 bits/s de idéntica manera (reiniciando la placa y volviendo a ejecutar su gestor de arranque) aunque ambas placas incorporen modelos de microcontrolador completamente diferentes al ATmega32U4.

Arduino Yún

Esta placa incorpora el mismo microcontrolador (con el mismo bootloader pregrabado y funcionando a los mismos 16MHz) que el modelo Micro (es decir, el ATmega32U4). Asimismo, ofrece 20 pines de entrada/salida digitales (7 de los cuales pueden actuar como salidas PWM) con una corriente máxima admitida de 40mA, 12 entradas analógicas de 10 bits de resolución, un conector ICSP (que se utilizará sobre todo como interfaz SPI), dos pines (los nº2 –SDA– y nº3 –SCL–) específicamente reservados para comunicación I²C y un zócalo USB micro-B a través del cual se recibe la alimentación eléctrica (que ha de venir ya regulada a su tensión de trabajo –5V–), se cargan los programas en la memoria Flash del ATmega32U4 y se establece la comunicación serie con el computador allí conectado.

NOTA: Fijarse que la placa Yún, a diferencia de la mayoría de otros modelos, no incorpora

EL MUNDO GENUINO-ARDUINO

ningún zócalo 5,5/2,1mm para recibir alimentación eléctrica, por lo que la única manera de conseguirla es mediante la conexión USB (o, si la fuente externa está regulada a 5V, a través de los pines "Vin" y "GND"). Afortunadamente, esta carencia la podremos solventar utilizando la combinación del producto nº **12889** de Sparkfun (o nº **276** de Adafruit, que es el mismo) con el producto nº **2727** de Adafruit o bien directamente usando solo el producto nº **12890** de Sparkfun (o el nº **1995** de Adafruit, que es el mismo). Otra solución alternativa podría ser emplear un módulo PoE, como el AG9700-FL fabricado por Silvertel, el cual proporciona 9W a 5V (de los módulos PoE hablaremos posteriormente, en el apartado correspondiente al shield Ethernet).

No obstante, la placa Yún añade una novedad muy importante que la distingue de todo el resto de placas Arduino: además de incorporar el chip ATmega32U4, también incluye un segundo chip que trabaja de forma independiente, el microprocesador AR9331 del fabricante Atheros (<https://www.qualcomm.com>). Que el AR9331 sea un "microprocesador" (en realidad lo es un componente en su interior llamado 24K, con arquitectura MIPS de 32 bits funcionando a 400MHz y 3,3V, fabricado por Imagination, <http://imgtec.com>) en vez de un "microcontrolador" implica (entre muchas otras consideraciones a nivel técnico en las que no vamos a entrar) que es capaz de ejecutar un sistema operativo completo (guardado dentro de su propia memoria Flash de 16MBytes y cargado en cada arranque en una memoria RAM propia de 64MBytes –de tipo DDR2–). De hecho, el AR9331 que se comercializa formando parte de la placa Yún viene con una distribución de Linux ya preinstalada, concretamente una variante mantenida por el equipo de Arduino llamada OpenWrt-Yún (<https://github.com/arduino/openwrt-yun>), la cual deriva de OpenWrt (<https://openwrt.org>), un sistema Linux muy extendido como sistema base de "routers" y dispositivos de conectividad de red en general.

NOTA: De forma similar a como ocurre con los microcontroladores, el microprocesador AR9331 también incorpora un bootloader que se encarga, en este caso, de gestionar el proceso de arranque del sistema OpenWrt-Yún. Este bootloader se llama U-Boot (<http://www.denx.de/wiki/U-Boot>) y es un programa que no tendremos por qué manipular nunca directamente.

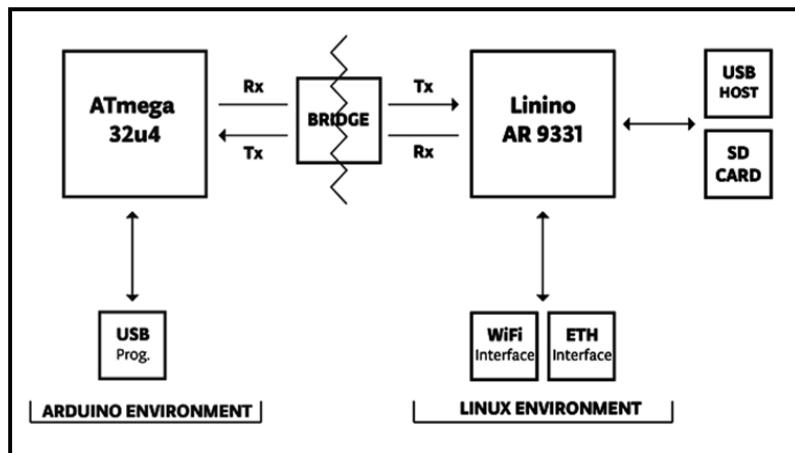
Los desarrolladores de Arduino han optado por usar este sistema operativo porque es ideal para el tipo de hardware que representa el microprocesador AR9331: este chip es capaz de realizar hasta 5 conexiones simultáneas a redes cableadas Ethernet de tipo 10/100Mbit/s (con posibilidad de emplear PoE si fuera necesario); es capaz de conectarse a redes inalámbricas WiFi de tipo IEEE 802.11b/g/n, ya sea como simple cliente o como punto de acceso (es decir, comportándose como un "router" doméstico; en este modo –llamado "modo AP", de "Access Point"– la placa no puede conectarse a Internet pero puede utilizarse como punto central al que se le pueden conectar otros dispositivos, como por ejemplo sensores inalámbricos), pudiendo estar estas redes WiFi aseguradas con los algoritmos criptográficos más habituales (WEP, WPA, WPA2-Personal o WPA2-Enterprise, entre otros); es capaz de

gestionar almacenamiento de datos (páginas web, scripts, etc.) en tarjetas de tipo SD (de ahí la presencia del zócalo microSD en la placa) y es capaz, finalmente, de actuar como "host" USB 2.0 (de ahí la presencia del conector USB de tipo A en la placa, al cual podremos enchufar los dispositivos "periféricos" deseados: "webcams", lápices USB, teclados, ratones, joysticks, etc., siempre y cuando el sistema OpenWrt-Yún disponga de los "drivers" adecuados para su correcto reconocimiento).

NOTA: Observar que, tras lo descrito en el párrafo anterior, que el modelo Yún *jes* la única placa Arduino que incorpora conectividad cableada e inalámbrica todo en uno!

La placa Yún ofrece, pues, un entorno donde dos chips diferentes trabajan en paralelo (pero comunicándose entre sí cuando es necesario a través de una conexión serie interna establecida entre ambos). Básicamente, su esquema de funcionamiento sería el siguiente: mientras que el chip ATmega32U4 se encarga de conectar con nuestro computador a través de su zócalo microUSB (para alimentarse, para establecer comunicación serie o para recibir la carga de un nuevo programa) y, sobre todo, se encarga de ejecutar dicho programa de forma similar a cualquier otra placa Arduino (pudiendo hacer uso, por tanto –y como es habitual–, de los GPIOs de la placa), el chip AR9331, a través del sistema OpenWrt-Yún, se encarga en exclusiva de la gestión de la conectividad Ethernet y WiFi, de la gestión del almacenamiento en la tarjeta microSD y del control del puerto USB de tipo A. Esto significa que en el código Arduino ejecutado por el ATmega32U4 no podremos utilizar las librerías oficiales "Ethernet", "WiFi101", "SD" ni "USBHost", ya que el hardware correspondiente no está accesible desde el ATmega32U4: se ha de gestionar obligatoriamente desde el sistema OpenWrt-Yún. A continuación se muestra un esquema (extraído de la propia web de Arduino) que ilustra esta "separación de responsabilidades" más claramente:

NOTA: "Linino" es el nombre original de lo que ha pasado a llamarse OpenWrt-Yún.



EL MUNDO GENUINO-ARDUINO

Ya hemos mencionado que la conexión entre ambos chips se establece mediante un canal interno de tipo serie. Un extremo de este canal son los pines 0 (RX) y 1 (TX), los cuales están conectados al controlador UART del ATmega32U4 (y son gestionables desde el código Arduino mediante el objeto *Serial1*) y el otro es el puerto serie hardware del AR9331 (disponible bajo la carpeta /dev del sistema Linux como un dispositivo más -concretamente, el dispositivo /dev/ttym0-). No obstante, tal como señala el esquema anterior, existe una manera más sencilla de comunicar ambos chips, que es mediante una librería oficial del lenguaje Arduino (y por tanto, utilizable en los códigos que escribiremos para ser ejecutados por el ATmega32u4) llamada "Bridge" (<https://www.arduino.cc/en/Reference/YunBridgeLibrary>). Esta librería permite que nuestro código Arduino sea capaz –entre otras cosas– de poner en marcha cualquier ejecutable instalado en el sistema OpenWrt-Yún (pudiendo, opcionalmente, pasárle valores –obtenidos de sensores, por ejemplo– y obteniendo como retorno el resultado de dicha ejecución, si lo hubiera); en este sentido, controlar el puerto USB "host" o conectar con alguna red cableada o inalámbrica para acceder al exterior (y así enviar peticiones y recibir e interpretar las respuestas correspondientes) son solo dos de las múltiples posibilidades que muchos programas disponibles en OpenWrt-Yún ofrecen, pero la acción concreta a realizar dependerá del ejecutable invocado. Otra funcionalidad interesante que la librería Bridge también permite añadir a nuestro código Arduino es la capacidad de leer/modificar/crear/borrar directamente ficheros y carpetas almacenadas en la tarjeta microSD. Y otra también es facilitar muchísimo la implementación en nuestro código Arduino de clientes web y/o servidores web cuya conexión al exterior se produce a través del AR9331.

NOTA 1: Para que la librería Bridge reconozca el contenido alojado en la tarjeta microSD, este ha de estar guardado dentro de una carpeta llamada "arduino" ubicada dentro de la raíz de la tarjeta (raíz que el sistema OpenWrt asume que es la carpeta "/mnt/sda1"). La tarjeta, además, deberá estar formateada en FAT32, NTFS, EXT3/4 o HFS(+).

NOTA 2: Si se utiliza la funcionalidad de servidor web que ofrece la librería Bridge, hay que saber que todo el contenido guardado dentro de una carpeta llamada "arduino/www" ubicada en la raíz de una tarjeta microSD (esto es, accesible dentro del sistema OpenWrt-Yún como "/mnt/sda1/arduino/www") es automáticamente compartido vía web en la dirección http://IP_placa_yún/sd.

NOTA 3: La contrapartida de la librería Bridge en el lado del AR9331 es un script Python llamado "bridge.py" (cuya existencia no tendríamos por qué conocer a no ser que quisieramos modificarlo para alterar el funcionamiento predeterminado de la placa Yún).

Al ser OpenWrt-Yún un sistema operativo completo, de entrada ya viene con muchos programas preinstalados habituales en cualquier distribución Linux que esté basada en el terminal de comandos (como por ejemplo la aplicación *curl* o el intérprete *python*, por citar un par). De todas formas, si así lo necesitáramos también

tenemos la posibilidad de descargar e instalar una gran variedad de aplicaciones de todo tipo para añadir funcionalidad extra al sistema base; de esta manera podremos implementar todo tipo de servidores (un ejemplo típico sería, por nombrar alguno, un servidor de streaming de vídeo captado por una webcam en tiempo real) así como también desarrollar nuestros propios programas escritos en lenguajes inicialmente no disponibles en la instalación base (como Node.js, etc.). Para instalar software adicional, la opción más rápida es entrar en el terminal del sistema OpenWrt-Yún (ya sea vía SSH o bien a través del "Serial Monitor" del entorno Arduino; la manera concreta será detallada en los siguientes párrafos) y allí dentro ejecutar el programa *opkg* con las opciones deseadas: *update*, *install*, *upgrade*, *remove*... Este programa es el "gestor de paquetes" del sistema OpenWrt; para obtener más información sobre su uso y posibles opciones, el lector puede consultar la página <https://www.arduino.cc/en/Tutorial/YunPackageManager>. Para conocer la lista completa de los paquetes disponibles para instalar, se puede consultar el enlace <https://github.com/arduino/openwrt-packages-yun>.

La placa Yún también permite ser programada, además de mediante el cable USB (como es habitual), a través de WiFi (gracias a que el AR9331 es capaz de actuar como programador ISP del ATmega32U4 a través de un canal SPI interno compartido entre ambos chips). De esta manera nos será muy cómodo cargar nuevos programas en placas que estén situadas a varios metros de distancia de nuestro computador. Para ello, no obstante, tendremos que modificar el modo de trabajo que tiene la placa por defecto (que es el "modo AP") al otro modo posible (el "modo cliente"); de esta manera la placa pasará a conectarse al mismo AP al que está conectado nuestro computador (que, en entornos domésticos, suele ser nuestro "router" de casa) como un elemento más de la WiFi compartida. Los pasos para realizar esta configuración se detallan en la siguiente "Breve nota":

Breve nota sobre cómo conseguir cargar programas en la placa Yún vía WiFi

Si no se ha tocado su configuración de fábrica, cada vez que se enciende una placa Yún, debido a estar en "modo AP", genera una red WiFi llamada "ArduinoYún-XXXXXX" (donde las equis indican la dirección MAC del módulo WiFi). Tras seleccionar dicha red WiFi en nuestro computador; este debería recibir una determinada dirección IP de parte de la placa Yún para poder empezar a comunicarse con ella. Si este paso ha funcionado, seguidamente deberemos abrir un navegador y escribir en su barra de direcciones o bien <http://arduino.local> o bien <http://192.168.240.1> (es lo mismo): veremos una página web preguntando por una contraseña, la cual por defecto es "arduino".

La página que aparece a continuación muestra información de diagnóstico sobre

EL MUNDO GENUINO-ARDUINO

las conexiones de red actuales (concretamente, arriba muestra ciertos datos sobre la interfaz WiFi del AR9331 –llamada "WLAN0" por OpenWrt– y abajo sobre la interfaz Ethernet –llamada "ETH1" por OpenWrt–). Tras hacer clic sobre el botón "Configuration", en la nueva página que se nos muestre podremos indicar por fin los datos necesarios para reconfigurar la placa. Los más importantes son:

YÚN NAME: es el nombre que deberemos escribir (seguido de ".local") en la barra del navegador cada vez que queramos acceder a este panel web de configuración (por defecto es "arduino").

PASSWORD: es la contraseña que deberemos escribir en la página inicial para poder acceder a este panel web de configuración (por defecto es "arduino").

CONFIGURE A WIRELESS NETWORK: esta opción ha de estar marcada para que las opciones siguientes tengan efecto.

DETECTED WIRELESS NETWORKS: en este desplegable podremos elegir la red WiFi detectada a la que queremos que se conecte la placa Yún como cliente. De forma alternativa, en vez de seleccionar esa red de la lista podemos escribir su nombre en el cuadro WIRELESS NAME.

SECURITY: en este desplegable elegiremos el tipo de seguridad (WEP, WPA, WPA2, etc.) que emplea la red WiFi seleccionada en la opción anterior. Según el valor seleccionado, deberemos introducir la clave asociada a la red WiFi segura elegida.

Una vez indicados los valores deseados en las opciones anteriores, solo falta hacer clic en el botón "Configure & Restart": esto reiniciará el chip AR9331 de la placa Yún con la nueva configuración ya activa (hasta nuevo cambio) incluyendo una nueva dirección IP, asignada por el AP responsable de la red WiFi elegida (el cual será compartido a partir de ahora por computador y placa). A partir de ese momento, y tras revincular nuestro computador de nuevo a ese AP, ya podremos programar el ATmega32U4 simplemente seleccionando en el menú *Tools->Port* del entorno software de Arduino la opción que muestra el nombre de la placa Yún junto a su dirección IP recién recibida del AP (en vez del puerto USB) y realizar a partir de entonces los pasos habituales.

Hay que tener en cuenta, no obstante, que cada vez que se vaya a cargar un programa de esta manera, el entorno preguntará la contraseña introducida en la opción PASSWORD mencionada anteriormente.

El panel web de control estudiado en la "Breve nota" anterior es el lugar donde más a menudo realizaremos los cambios de configuración que en algún momento necesitemos aplicar a la placa Yún, ya que nos ofrece un entorno muy sencillo y amigable, evitándonos el uso de (a veces complicados) comandos de terminal. Así pues, tareas habituales como instalar y desinstalar programas,

configurar las interfaces de red y los programas de arranque así como la supervisión del rendimiento y la revisión de los mensajes de error se convierten en algo muy sencillo de llevar a cabo. Este panel web (que como software libre que es, tiene su código disponible en <https://github.com/arduino/YunWebUI>) en realidad es una variante del panel web oficial del proyecto OpenWrt, el cual se llama LuCI y está disponible en <https://github.com/openwrt/luci>.

El panel web no ofrece, desgraciadamente, todas las posibilidades de configuración que el sistema OpenWrt admite. Así que en alguna ocasión puede que sea necesario acceder a un terminal del OpenWrt para ejecutar los comandos adecuados directamente allí de forma interactiva. Esto es posible conseguirlo de dos maneras. Una de ellas es utilizando un cliente SSH (programa que ha de instalar aparte en nuestro computador) y otra es utilizando el "Serial Monitor" que viene dentro del entorno software de Arduino.

SSH ("Secure Shell") es un protocolo que sirve para acceder a máquinas remotas (en este caso, al sistema OpenWrt de la placa Yún desde el sistema de nuestro computador) a través de una red. Por tanto, la placa Yún deberá estar accesible, ya sea a través de WiFi o a través de cable Ethernet, y además deberemos conocer, en cualquier caso, la dirección IP que ha recibido de nuestro AP (hay que saber, en este sentido, que si la placa Yún se conecta mediante cable, esta está configurada de fábrica para solicitar una dirección IP automáticamente vía DHCP: la IP asignada se puede conocer observando el menú *Tools->Port* del entorno Arduino). SSH permite manejar por completo el sistema remoto a través de un terminal y además, es seguro porque utiliza técnicas de cifrado que hacen que la información que viaja por el medio (aire o cable) se transfiera de manera no legible, evitando que terceras personas puedan descubrir el usuario y contraseña de la sesión. Para poder comunicarnos mediante SSH será necesario, tal como ya hemos dicho, un software específico (un "cliente SSH") instalado en nuestro computador; en sistemas Linux y OS X ya viene uno predefinido formando parte del sistema estándar (el cual se ha de invocar desde un terminal de nuestro computador) pero en Windows lo deberemos conseguir aparte (en este sentido, un cliente SSH gratuito y fiable es la aplicación Putty, <http://www.putty.org>). En cualquier caso, el usuario que hay que indicar para acceder al sistema OpenWrt es "root" y la contraseña es aquella que se especificó en la opción PASSWORD mencionada en la "Breve nota" anterior (su valor por defecto, si no se cambió, es "arduino").

La otra manera de entrar en un terminal interactivo del sistema OpenWrt es mediante el "Serial Monitor" que viene incorporado dentro del entorno de Arduino, el cual permite aprovechar la conexión USB entre placa y computador, sin necesidad

EL MUNDO GENUINO-ARDUINO

de emplear ni Ethernet ni WiFi. Para que esto funcione, no obstante, primero deberemos cargar en el ATmega32U4 (de la forma habitual) un determinado programa Arduino ya preparado llamado "YunSerialTerminal", disponible bajo el menú "File->Examples->Bridge" del entorno Arduino. Una vez realizada esta carga, ya sí podremos abrir el "Serial Monitor" y, tras seleccionar el valor "New Line" del cuadro desplegable visible en la zona inferior de la ventana, aparecerá ante nosotros el terminal interactivo deseado desde donde podemos ejecutar los comandos del sistema OpenWrt necesarios.

Queda fuera de los objetivos de este libro profundizar en los múltiples comandos que podríamos llegar a utilizar en un terminal interactivo como el descrito en párrafos anteriores. Tal solo comentaremos que en el sistema OpenWrt existe un comando propio (es decir, no existente en otros tipos de Linux) llamado *uci* que permite realizar cambios de una forma muy estructurada, coherente y homogénea en la configuración del sistema (la cual está basada muchas veces en diferentes ficheros con distinto formato esparcidos por varias carpetas). De hecho, este comando *uci* es la base "invisible" sobre la que se sustenta la funcionalidad del panel web LuCI ya conocido. Sus posibilidades y características se pueden consultar en su documentación oficial, <http://wiki.openwrt.org/doc/uci>

El sistema OpenWrt-Yún ocupa alrededor de 9MBytes de los 16Mbytes disponibles en la memoria Flash interna del AR9331. Afortunadamente, es posible utilizar una tarjeta microSD (que no deja de ser una memoria Flash de muchísima más capacidad) para alojar el sistema completo, obviando entonces la existencia de la memoria Flash del AR9331. De esta manera, en vez de tener el sistema en una memoria (la Flash interna) y los eventuales datos en otra (la tarjeta microSD, hasta ahora opcional) pasariamos a tener tanto el sistema como los datos en un solo lugar: la tarjeta microSD (la cual, eso sí, podría haberse particionado antes para separar ambos tipos de contenido). El procedimiento exacto aparece detallado paso a paso en el enlace <https://www.arduino.cc/en/Tutorial/ExpandingYunDiskSpace>

Otra tarea de mantenimiento que es posible que debamos/queramos hacer algún día es actualizar el sistema OpenWrt-Yún por completo (y así disfrutar de las mejoras, ampliaciones de funcionalidad y correcciones de errores introducidas en las nuevas versiones). Para poder realizar el procedimiento de actualización de una forma sencilla y rápida, el equipo de desarrolladores de Arduino ofrece en la página de descargas oficial (<https://www.arduino.cc/en/Main/Software>) un fichero .zip (señalado como "OpenWrt-Yún 1.X.X Upgrade Image") que, tal como indica su nombre, se corresponde –una vez descomprimido, ojo– con el sistema OpenWrt-Yún más actual hasta la fecha. El objetivo de la existencia de este fichero .zip es reducir la

complejidad de la actualización del sistema convirtiendo este proceso en una simple sobrescritura del sistema actual por el contenido –descomprimido– de ese fichero .zip. El procedimiento exacto, no obstante, se encuentra detallado en el enlace <https://www.arduino.cc/en/Tutorial/YunSysupgrade>.

NOTA: Para seguir los pasos marcados en el enlace anterior, nuestro computador ha de disponer de un zócalo de tarjetas microSD integrado: si no es el caso, una opción es adquirir un adaptador microSD<->USB como el producto nº 939 de Adafruit o el nº 13004 de Sparkfun.

Para acabar este apartado, comentaremos la utilidad de los tres botones de reseteo presentes en la placa Yún. El botón etiquetado como "Yún RST" pone a LOW la línea "reset" del AR9331, reiniciándolo (y, por tanto, reiniciando el sistema OpenWrt-Yún); esto significa que todos los datos presentes en ese momento en su memoria RAM se perderán (si no han sido previamente guardados en la memoria Flash o SD) y todos los programas ejecutándose en ese momento se interrumpirán. El botón etiquetado como "32U4 RST" pone a LOW la línea "reset" del ATmega32U4, reiniciándolo (y, por tanto, abriendo la posibilidad de proceder, al ejecutarse el bootloader, a la eventual carga de un nuevo programa; no obstante, este botón no se suele utilizar mucho porque el mismo reinicio se puede realizar directamente desde el entorno software Arduino, lo que resulta más cómodo). Finalmente, el botón etiquetado como "WLAN RST" tiene tres funciones: si se pulsa durante menos de 5 segundos, simplemente reinicia el módulo WiFi; si se pulsa entre 5 y 30 segundos, vuelve a establecer los valores por defecto "de fábrica" para el módulo WiFi (es decir, vuelve a poner la placa en "modo AP" con dirección IP fija 192.168.240.1 y con contraseña "arduino"; atención porque en este caso el AR9331 es reiniciado), y si se pulsa durante más de 30 segundos, además de hacer lo anterior, resetea completamente el sistema OpenWrt-Yún, borrando todos los ficheros y programas que no pertenecen a la instalación estándar y restableciendo los posibles cambios realizados en los ficheros de configuración (y, por tanto, provocando que la placa aparente haber sido recién adquirida), siempre y cuando, eso sí, se esté usando la memoria Flash interna del AR9331.

Arduino Lilypad, Lilypad Simple, Lilypad SimpleSnap y LilypadUSB

La placas Arduino de la familia LilyPad están diseñadas para ser cosidas a material textil. Si a ellas les conectamos (mediante hilos conductores) fuentes de alimentación, sensores y actuadores, obtendremos un circuito que puede "llevarse encima", haciendo posible, en definitiva, la creación de vestidos y ropa "inteligente" (lo que viene a llamar "wearables"). Además, estas placas se pueden lavar.

Todas las placas de la familia Lilypad excepto la LilypadUSB incorporan el microcontrolador ATmega328V (una versión de bajo consumo del Atmega328P); la

EL MUNDO GENUINO-ARDUINO

LilypadUSB, por su parte, se basa en el microcontrolador Atmega32U4, lo que le permite comportarse de la misma forma que la ya descrita al hablar del modelo Micro. Asimismo, todas las placas de la familia Lilypad, excepto la LilypadUSB, se pueden programar, al igual que el modelo Pro, mediante un adaptador o cable USB<->Serie (o –pero solo en el caso de la placa original, llamada "Main"– incorporándole un conector ICSP a los agujeros que ofrece para ello, bien sea soldándolo, bien mediante "pogo pins"); el modelo LilypadUSB, por su parte, incorpora un zócalo micro-USB que permite su programación directa vía USB.

Las placas LilypadUSB y Lilypad Simple son, por otro lado, las únicas de la familia que pueden ser alimentadas (mediante una batería LiPo de 3,7V) a través del zócalo JST que llevan incorporado, batería que, además, puede ser recargada mediante la conexión USB gracias a la presencia en la placa del chip cargador MCP73831. La placa Lilypad SimpleSnap ya lleva incorporada una batería LiPo de fábrica (de hecho, esta es la única diferencia entre la placa Lilypad SimpleSnap y la placa Lilypad Simple junto con, eso sí, la sustitución de los agujeros THT por enganches –"snaps"– conductores, de ahí el nombre). En cualquier caso, todas las placas de la familia Lilypad pueden alimentarse además o bien vía USB, o bien a través de los pines "+" y "-" presentes en una zona de su perímetro.

Todas las placas de la familia Lilypad, excepto la "Main", llevan un interruptor encargado de cortar la alimentación en caso necesario, provenga esta donde provenga (USB, batería, etc.). El voltaje de entrada de todas las placas Lilypad (excepto el de la LilypadUSB) no está regulado, pero, afortunadamente, su voltaje de trabajo está dentro de un rango amplio: entre 2,5V y 5,5V. La placa LilypadUSB, por su parte, incorpora un regulador, el MIC5219, que adapta los 3,8V-5V de entrada admitidos a los 3,3V de trabajo de la circuitería.

El número de entradas y salidas varía ligeramente entre los modelos de la familia Lilypad, por lo que remito el lector a las tablas comparativas expuestas al final de este capítulo, donde se pueden observar estas diferencias numéricas de un solo vistazo.

En el enlace <http://www.sparkfun.com/categories/135> se pueden encontrar bastantes complementos interesantes adaptados en tamaño y flexibilidad a la familia LilyPad, como por ejemplo sensores de temperatura, luz y acelerómetros, LEDs de diferentes colores, motores de vibración, zumbadores, transmisores-receptores inalámbricos, portapilas para baterías LiPo, botón o AAA, breadboards, interruptores, botones, variantes específicas de la placa, etc. Incluso se distribuyen conjuntos de estos complementos en kits llamados "ProtoSnaps" para una mayor comodidad.

CAPÍTULO 2: HARDWARE GENUINO

Por otro lado, en la tienda oficial de Arduino (ver apéndice A) se puede adquirir el "Wearable Kit", formado por un conjunto de componentes (resistencias, potenciómetros, breadboards, hilo conductor...), un sensor de presión y actuadores varios (botones, LEDs...), todos ellos textiles. Este kit está diseñado por la empresa especializada Plug&Wear.

Para saber más sobre posibles usos y trucos de esta placa, se pueden consultar -entre otros- los tutoriales nº 281, nº 308, nº 312, nº 313 y nº 333 de Sparkfun, y también la muy completa y práctica web de su co-diseñador, Leah Buechley <http://lilypadarduino.org>. Si se desean conocer proyectos ya realizados con estos componentes, un buen lugar para inspirarse es <http://www.kobakant.at/DIY>

Arduino Gemma

Esta placa, diseñada en colaboración con Adafruit, es, al igual que los modelos Lilypad, de tipo "wearable" pero tiene un tamaño más reducido e incorpora un microcontrolador menos capaz, el ATtiny85. Este chip tan solo tiene 8Kbytes de memoria Flash (2,75 de los cuales son empleados por el bootloader), 512 bytes de SRAM y otros tantos de EEPROM y su número de entradas/salidas es muy reducido. NOTA: En realidad, el microcontrolador ATtiny85 pertenece a una subfamilia AVR diferente a la del resto de microcontroladores de 8 bits que hay en las distintas placas Arduino.

La placa Arduino Gemma contiene un zócalo JST para poder conectar allí una batería LiPo de 3,7V (su tensión de trabajo es de 3,3V regulados gracias al chip MIC5225), un interruptor para cortar la alimentación cuando sea necesario y se programa a través del conector microUSB que lleva incorporado.

Arduino Due

Esta placa pertenece a una familia totalmente distinta de la del resto de placas Arduino vistas hasta ahora porque incluye el microcontrolador SAM3X8E, el cual, aunque fabricado también por Atmel, es de una arquitectura interna muy diferente a la AVR (concretamente es de tipo ARM Cortex-M3); además, sus registros son cuatro veces más grandes de lo habitual en las otras placas (concretamente, son de 32 bits) y su velocidad de reloj está también muy por encima del resto de placas Arduino (concretamente, es de 84 MHz). Y para rematar, el microcontrolador SAM3X8E dispone de muchas más memoria (concretamente, 96KB de SRAM y 512KB de memoria Flash) así como también de un circuito especializado (llamado controlador "DMA") que permite a la CPU acceder a la memoria de una manera mucho más rápida.

EL MUNDO GENUINO-ARDUINO

Todo esto implica que con la placa Arduino Due se pueden hacer más cosas, y más rápidamente, por lo que permite ejecutar aplicaciones que realizan un gran procesado de datos.

Otros datos técnicos de la Arduino Due son: dispone de 54 pines de entrada/salida digital (12 de los cuales pueden ser usados como salidas PWM capaces de manejar hasta 12 bits de resolución), 12 entradas analógicas capaces de manejar hasta 12 bits de resolución, 4 chips TTL-UART (es decir, cuatro canales serie hardware independientes), 2 conversores digitales-analógicos (DAC) de 12 bits de resolución ubicados en los pines señalados como DAC0 y DAC1, 2 puertos I²C independientes (¡novedad!), 1 puerto SPI (el cual solo está implementado en los pines "ICSP"), 1 conector USB de tipo micro-B y otro de tipo micro-A, un zócalo de 2,1mm tipo "jack", un botón de reinicio y un botón de borrado. Además, ofrece como es habitual los pines "Vin", "GND", "5V", "3,3V", "IOREF", "AREF" y "Reset". Si el lector quiere conocer la vinculación existente entre cada pin-hembra de la placa Due y la patilla correspondiente del microcontrolador SAM3X-8E, puede consultar la tabla <https://www.arduino.cc/en/Hacking/PinMappingSAM3X>.

Un aspecto muy importante de esta placa que hay que saber es que su voltaje de trabajo es de 3,3V. Esto significa que la tensión máxima que los pines de entrada/salida pueden soportar es ese. Si se les proporciona un voltaje mayor (como los 5V a los que estamos habituados), la placa se podría dañar. No obstante, las fuentes de alimentación externas pueden ser las mismas que las utilizadas con la placa Arduino UNO, ya que sus rangos de tensión de entrada son idénticos (6-20V teóricos, 7-12V recomendables); esto es debido a que la tensión es adecuadamente reducida gracias a un regulador interno. Por otro lado, aunque la intensidad máxima que pueden ofrecer los pines "3,3" y "5V" es de 800mA, los pines-hembra de entrada y salida solo pueden resistir como mucho hasta 9mA y 15mA cada uno, respectivamente (y 130 mA en total); notar que dichos valores son bastante menores que los existentes en las placas basadas en el ATmega328 como la UNO, por ejemplo.

La Arduino Due mantiene la forma y disposición de la placa Arduino Mega, siendo compatible con todos los shields que respeten la misma disposición de pines y que, importante, trabajen a 3,3V. Por otro lado, todos los pines de entrada/salida tienen una resistencia "pull-up" interna desconectada por defecto de 100KΩ.

La Arduino Due ofrece dos conectores USB para separar dos funcionalidades diferentes. El conector micro-B más cercano al jack de alimentación (llamado "puerto de programación") está pensado para enchufar la placa al computador y transferir desde el entorno de desarrollo nuestro programa para que sea ejecutado por el microcontrolador, y a partir de allí mantener la comunicación serie entre computador

y placa. De hecho, este conector está controlado por el mismo chip ATmega16U2 que la placa Arduino UNO, por lo que su comportamiento es idéntico. El conector micro-BA más cercano al botón de reinicio (llamado "puerto nativo"), en cambio, está controlado directamente por el chip SAM3X8E y está pensado para usar la placa como un periférico USB más (como un teclado o un ratón, tal como también ocurre en las placas Micro o Yún). Pero además, una novedad que ofrece este último conector es que también permite a la placa actuar como "host USB". De esta forma, no solamente podríamos usar la placa Due como un teclado o un ratón, sino que podríamos conectarle a ella un teclado o un ratón reales, entre otros muchos dispositivos, como teléfonos móviles de última generación, por ejemplo.

NOTA: La capacidad que tienen algunos dispositivos (como la placa Arduino Due) de actuar tanto como periféricos USB como "hosts" USB indistintamente es una característica añadida al estándar USB 2.0 llamada USB "On-The-Go" (o, abreviadamente, USB OTG). Lógicamente, para que la comunicación OTG se pueda realizar correctamente, ambos extremos deben ser compatibles (ya que así, mientras uno actúa como "host" el otro actuará como periférico, y viceversa). Además, también necesitaremos un cable OTG para conectar ambos extremos, como los productos nº **11604** de Sparkfun (con conector micro-A) o nº **1099** de Adafruit (con conector micro-B).

La forma de programar la placa Arduino Due es similar a las anteriores placas, tanto en el uso del entorno de desarrollo como en el propio lenguaje de programación: todos los cambios están "bajo la superficie". El único detalle que hay que tener en cuenta es que la memoria Flash del microcontrolador ha de ser borrada "manualmente" cada vez que se le quiera cargar en nuestro programa. Esto es así porque el bootloader de esta placa está alojado en una memoria de tipo ROM separada de la memoria Flash, y solamente se ejecuta cuando detecta que la memoria Flash está vacía. De ahí la existencia del botón de borrado (marcado como "Erase") en la placa. Afortunadamente, si conectamos esta a nuestro computador mediante el "puerto de programación", este proceso de borrado de la memoria Flash se realiza de forma automática. En este sentido, la comunicación USB entre placa y computador se realiza de la misma forma en una la placa Due que en la placa UNO.

Arduino Zero

Esta placa incluye un microcontrolador fabricado por Atmel llamado SAM-D21(G18), el cual es de tipo ARM Cortex-M0+. Al igual que ocurría con el micro de la placa Arduino Due, el SAM-D21 tiene los registros de 32 bits pero, sin embargo, su velocidad de reloj es menor (concretamente es de 48MHz). La cantidad de memoria también es menor que la que ofrecía la placa Due (concretamente, el modelo Zero dispone de 32KB de SRAM y 256KB de memoria Flash).

EL MUNDO GENUINO-ARDUINO

Otros datos técnicos de la placa Arduino Zero son: dispone de 20 pines de entrada/salida digital (de los cuales todos excepto los nº 0, 1, 2 y 7 pueden ser usados como salidas PWM capaces de manejar hasta 12 bits de resolución), 6 entradas analógicas capaces de manejar hasta 12 bits de resolución, 1 chip TTL-UART (es decir, solo un canal serie hardware –disponible en los pines nº0 (RX) y nº1 (TX) –; realmente el micro SAM-D21 internamente dispone de más canales pero el entorno Arduino –por ahora– los ignora), 1 conversor digital-analógico (DAC) de 10 bits de resolución disponible en el pin A0, 1 reloj/calendario RTC, 1 puerto I²C disponible en los pines "SCL" y "SDA", 1 puerto SPI solo disponible en los pines "ICSP", 2 conectores USB de tipo micro-B, un zócalo de 2,1mm tipo "jack", un botón de reinicio y un botón de borrado. Además, ofrece como es habitual los pines "Vin", "GND", "5V", "3,3V", "IOREF", "AREF" y "Reset".

Un aspecto muy importante de esta placa que hay que saber es que su voltaje de trabajo es de 3,3V. Esto significa que la tensión máxima que los pines de entrada/salida pueden soportar es ese. Si se les proporciona un voltaje mayor (como por ejemplo 5V), la placa se podría dañar. No obstante, las fuentes de alimentación externas pueden ser las mismas que las utilizadas con la placa Arduino UNO, ya que sus rangos de tensión de entrada son idénticos (6-20V teóricos, 7-12V recomendables); esto es debido a que la tensión es adecuadamente reducida gracias a un regulador interno. Por otro lado, aunque la intensidad máxima que pueden ofrecer los pines "3,3" y "5V" es de 800mA, los pines-hembra de entrada/salida (los cuales tienen una resistencia "pull-up" interna –desconectada por defecto– de 50 KΩ) solo pueden resistir como mucho hasta 7mA de corriente cada uno (¡valor mucho menor del soportado por placas basadas en el microcontrolador ATmega328 como es el habitual modelo UNO!). Finalmente, conviene saber que esta placa mantiene la forma y disposición de la placa Arduino UNO, siendo compatible por tanto con todos los shields que respeten la misma disposición de pines y que, importante, trabajan a 3,3V (o que estén adecuadamente adaptadas).

La Arduino Zero ofrece dos conectores USB micro-B para separar dos funcionalidades diferentes. El conector más cercano al jack de alimentación (el "puerto de programación") está pensado para enchufar la placa al computador y (tal como haríamos con la placa UNO), transferir desde el entorno de desarrollo nuestro programa para que sea ejecutado por el microcontrolador, y a partir de allí, mantener la comunicación serie entre computador y placa; este conector está controlado por un chip AT32UC3A que implementa un firmware llamado EDGB, el cual es capaz de actuar, entre otras cosas, como conversor USB<->Serie (de forma análoga a como lo hacía el chip ATmega16U2 en la placa UNO). Por otro lado, el conector más cercano al botón de reinicio (el "puerto nativo") está controlado

directamente por el chip SAM-D21 y está pensado (igual que el "puerto nativo" de la placa Arduino Due) para permitir actuar a la placa como un periférico USB más (y simular así un funcionamiento de teclado, de ratón, de joystick, etc.) o como un "host" USB (pudiéndole entonces conectar muchos dispositivos reales, como un teclado, un ratón, un teléfono de última generación, etc.).

El firmware EDGB, además de ejercer como conversor USB<->Serie, también es capaz de funcionar como gestor de arranque para el micro SAM-D21 (usando un protocolo específico llamado "Serial Wire Debug, SWD") y también como depurador (o "debugger", en inglés). Esto último permite poder pausar en tiempo real la ejecución del código y poder así inspeccionar el estado y valor de las diferentes variables existentes en ese momento; a partir de esa pausa, el depurador permite realizar además otras acciones, como el poder avanzar la ejecución línea a línea (o función a función) e ir observando el comportamiento del programa, entre otras. Un depurador es, por tanto, una herramienta muy valiosa a la hora de encontrar posibles errores de código muy difíciles de localizar de otra manera. Desgraciadamente, el entorno Arduino oficial no permite hacer uso de las capacidades de depuración del firmware EDGB, por lo que para ello sería necesario utilizar un software distinto, como por ejemplo el entorno Atmel Studio (<http://www.atmel.com/tools/ATMELSTUDIO.aspx>) o directamente las herramientas sobre las que se basa –ambas ejecutables desde el terminal–: OpenOCD (<http://openocd.org>) y GDB (<http://www.gnu.org/software/gdb>).

Arduino 101

Esta placa está basada en un chip llamado Intel Curie, el cual a su vez contiene en su interior un microprocesador Intel Quark SE (cuya arquitectura es la x86, de 32 bits) funcionando a 32MHz y con un sistema operativo de tiempo real (un "RTOS") ya preinstalado llamado ViperOS, un giroscopio y un acelerómetro de 6 ejes, un módulo que aporta conectividad Bluetooth Smart (también conocida como BLE, de "Bluetooth Low Energy" o "Bluetooth 4.0") y un chip DSP ("Digital Signal Processor") capaz de medir, filtrar y procesar señales analógicas provenientes de múltiples sensores. No obstante, a día de hoy no se han publicado aún las librerías oficiales que permiten aprovechar estas funcionalidades hardware.

Dispone asimismo de 384KBytes de memoria Flash y de 80KBytes de SRAM (aunque solo 24KBytes están realmente disponibles para nuestros programas), de 14 salidas/entradas digitales (4 de ellas pudiendo actuar como PWM), de 6 entradas analógicas, de un zócalo 5,5/2,1mm para la alimentación, de un zócalo USB de tipo B (usado tanto para la comunicación serie con el computador como para la carga de nuestro programa en la placa), de un conector ICSP (utilizable como –único– zócalo

EL MUNDO GENUINO-ARDUINO

SPI) y de un par de pines-hembra exclusivo para la comunicación I²C convenientemente señalados. Su tensión de trabajo (regulada a partir de una tensión de entrada de entre 7V y 12V) es de 3,3V pero sus pines soportan 5V; lo que no soportan es una corriente mayor de 4mA.

Tablas comparativas de los diferentes modelos de placas

Para que el lector se haga una idea más concisa de las diferentes capacidades que ofrecen los modelos de placas Arduino comentados en los apartados anteriores, a continuación ponemos a su disposición, a modo de resumen, una serie de tablas de rápida consulta en las que se muestran algunas de sus características más relevantes.

La primera tabla muestra los modelos actuales de placas Arduino que contienen un microcontrolador de tipo AVR (8 bits). A destacar (como datos que no aparecen en la tabla) que el modelo Yún es el único que carece de zócalo 5,5/2,1mm para recibir alimentación y que el modelo Pro es el único que dispone de zócalo JST para poder conectar allí una batería LiPo externa.

Modelo	UNO	Nano	Pro Pro Mini	Mega	Micro	Yún
Micro		ATmega328P		ATmega2560	ATmega32U4	Atheros AR9331
Arq. micro				AVR 8 bits (ATmega)		MIPS 24K
V. trabajo (V)	5	3,3/5		5		3,3
V. entrada (V)	7-12	3,3/5-12		7-12		5
Freq. (MHz)	16	8/16		16		400
Nº ent. analóg.	6	8	6	16		12
Nº sal. analóg.				0		
Nº pines digit.		14		54		20
Nº pines PWM			6	15		7
I máx. en pin				40mA		
Flash (KB)		32		256	32	64MB
SRAM (KB)		2		8	2,5	16MB
EEPROM (KB)		1		4	1	0
Conektor USB	B	Mini	-	B		Micro
SPI					1	
I ² C					1	
Objetos Serial		1		4		2

La siguiente tabla muestra los modelos de placas Arduino específicamente diseñados para ser cosidos a ropa para formar circuitos "wearables". A destacar (como dato que no aparecen en la tabla) que el único modelo que carece de zócalo JST para conectar allí una batería LiPo externa es el modelo "Lilypad" (y el "SimpleSnap", pero este ya lleva la batería incorporada):

CAPÍTULO 2: HARDWARE GENUINO

Modelo	Lilypad	Lilypad Simple	Lilypad SimpleSnap	Lilypad USB	Gemma			
Micro	ATmega328V		ATmega328	ATmega32U4	ATtiny85			
Arq. micro	AVR 8 bits (ATmega)			AVR 8 bits (ATTiny)				
V. trabajo (V)	2,7-5,5		3,3					
V. entrada (V)	2,7-5,5		3,8-5	4-16				
Freq. (MHz)	8							
Nº ent. analóg.	6		4		1			
Nº sal. analóg.	0							
Nº pines digit.	14		9		3			
Nº pines PWM	6		5	4	2			
I máx. en pin	40mA							
Flash (KB)	16		32		8			
SRAM (KB)	1		2	2,5	0,5			
EEPROM (KB)	0,5		1		0,5			
Conector USB	-		Micro					
SPI	0							
I²C	0							
Objetos Serial	1							

La siguiente tabla muestra, finalmente, los modelos de placas Arduino que contienen un microcontrolador de 32 bits (ya sea de tipo ARM o bien x86):

Modelo	Due	Zero	101
Micro	SAM3X(8E)	SAMD21(G18)	Curie (Quark SE)
Arq. micro	ARM Cortex-M3	ARM Cortex-M0+	x86
V. trabajo (V)	3,3		
V. entrada (V)	7-12		
Freq. (MHz)	84	48	32
Nº ent. analóg.	12 (12-bit)		6 (12-bit)
Nº sal. analóg.	2 (12-bit)	1 (10-bit)	0
Nº pines digit.	54	20	14
Nº pines PWM	12 (12-bit)	18 (12-bit)	4 (12-bit)
I máx. en pin	15mA	7mA	4mA
Flash (KB)	512	256	196
SRAM (KB)	96	32	24
EEPROM (KB)	0	0	0
Conector USB	Micro (2)		B
SPI	1		
I²C	2		1
Objetos Serial	5	3	1

EL MUNDO GENUINO-ARDUINO

Por otro lado, conviene saber que todos los modelos de placas anteriores (y todos los shields oficiales también) cumplen con los estándares industriales impuestos tanto por la FCC como por la UE en relación con la regulación del espectro electromagnético. Lo explicamos en la siguiente nota.

Breve nota sobre las regulaciones del espectro electromagnético

Todos los modelos de placa Arduino están certificados tanto por la Comisión Federal de Comunicaciones estadounidense (FCC) como por la Unión Europea. Eso significa que Arduino cumple en ambas zonas geográficas con las normativas pertinente en el ámbito de las emisiones electromagnéticas de aparatos electrónicos y de telecomunicaciones. Estas normativas son necesarias para evitar que este tipo de señales (entre las que se encuentran las de telefonía móvil, las WiFi, las Bluetooth, las infrarrojas, etc.) emitidas por diferentes aparatos interfieran entre sí generando el caos; para ello dichas reglamentaciones distribuyen el espectro electromagnético en varias franjas de frecuencia y a cada una le asignan un uso concreto y regulado mediante licencias.

Podemos comprobar que efectivamente Arduino respeta las normas de ambos organismos observando sus respectivos logos en el dorso de la placa. La importancia de tener estos logos impresos es facilitar la aceptación de las placas Arduino en su acceso a los distintos mercados (estadounidense y europeo según el caso), ya que si no estuvieran certificados, su entrada podría ser rechazada al no cumplir la normativa legal.

¿QUÉ "SHIELDS" ARDUINO OFICIALES EXISTEN?

Además de las placas Arduino propiamente dichas, también existen los llamados "shields". Un "shield" (en inglés significa "escudo") no es más que una placa de circuito impreso que se coloca en la parte superior de una placa Arduino y se conecta a ella introduciendo las ristras de pines-macho sobresalientes en su dorso dentro de las ristras de pines-hembra que la placa Arduino en cuestión ofrece. Con este sistema, por tanto, no es necesario utilizar ningún cable para conectar la placa Arduino con la PCB que representa ser todo shield. Dependiendo del modelo, incluso sería posible apilar varios shields uno encima de otro, pero esto dependerá de si el shield inferior ofrece pines-hembra para poder acoplarlos a su vez a los pines sobresalientes del dorso del shield superior.

La función de los shields es actuar como placas supletorias, ampliando las capacidades y complementando la funcionalidad de la placa Arduino base de una

forma más compacta y estable. Una vez acoplado un shield sobre una placa Arduino, a partir de entonces serán los GPIOs ofrecidas por este shield (y también sus pines de alimentación como GND, 5V o 3V3-, AREF o RESET) las que utilizaremos en nuestros circuitos. En este sentido, los pines del shield directamente encajados sobre los pines-hembra de la placa Arduino subyacente tienen exactamente la misma disposición y funcionalidad que estos últimos.

No obstante, hay que saber que los shields suelen monopolizar el uso de algunos pines de entrada/salida para su propia comunicación con la placa subyacente, por lo que estos quedan "inutilizados" para cualquier otro uso. Por ejemplo, si se acoplan varios shields uno sobre otro y todos se comunican mediante SPI con la placa Arduino, todos ellos podrán usar sin problemas los pines comunes correspondientes a las líneas MISO, MOSI y SCLK, pero cada uno de ellos deberá usar un pin diferente como línea CS. Estos detalles deben conocerse en la documentación de cada shield.

Por otro lado, también hay que tener en cuenta los requerimientos de alimentación eléctrica que necesitan los shields. Ya sabemos que, por ejemplo, mediante una conexión USB una placa Arduino puede llegar a recibir alrededor de 500mA, por lo que la corriente que queda para el funcionamiento de un posible shield en ese caso es pequeña. Tipos de shields que consumen mucho (de hasta 300mA) son los que tienen pantallas LCD o los que proporcionan conectividad Wi-Fi, por ejemplo. Otro dato importante que hay que tener en cuenta también es saber si la tensión de trabajo de un shield (5V o 3,3V) es compatible con la tensión de trabajo de la placa subyacente.

Existen literalmente centenares de shields construidos por la comunidad compatibles con la placa UNO que le aportan un plus de versatilidad, pero "shields" oficiales solo existen los siguientes:

Arduino Ethernet Shield

Este shield está pensado para los que le quieran añadir a las placas Arduino compatibles en forma (UNO, Mega, Due, Zero...) la capacidad de conectarse a una red cableada TCP/IP. Esto se consigue gracias al chip controlador W5100 que lleva incorporado este shield (del fabricante Wiznet, <http://www.shopwiznet.com>), chip que deberemos configurar en el código de nuestro programa usando una librería de programación llamada "Ethernet", la cual ya forma parte por defecto del lenguaje Arduino oficial.

EL MUNDO GENUINO-ARDUINO

Este shield requiere 5V para funcionar (voltaje aportado por la placa subyacente mediante el encaje del pin de alimentación correspondiente entre placa y shield) pero es compatible con las placas cuya tensión de trabajo es de 3,3V (como la Due o la Zero) gracias al elevador DC/DC que este shield lleva incorporado.

El procedimiento de cargar nuestros programas en el microcontrolador de la placa Arduino acoplada al shield Ethernet no varía respecto al realizado normalmente con una placa Arduino independiente: primero debemos conectar la placa Arduino a nuestro computador mediante el cable USB, y una vez cargado el programa, como siempre, podremos seguir alimentando la placa vía USB o bien desconectarla del computador y enchufarla a una fuente de alimentación externa. A partir de entonces, al conector RJ-45 que lleva incorporado el shield Ethernet le podremos conectar un cable de red (técticamente, un cable de par trenzado de categoría 5 o 6) de tipo "estándar" para comunicar el shield con un "switch" o un "router" o de tipo "cruzado" si deseamos comunicar el shield directamente a un computador.

Breve nota sobre Ethernet

El nombre "Ethernet" sirve para definir un determinado tipo de red cableada diseñada para el tránsito de datos (y, eventualmente, de alimentación eléctrica) entre los distintos dispositivos que estén conectados a ella. En concreto, para que una red sea de "tipo Ethernet", las características de su cableado y de las señales y tramas de datos que viajan por él han de seguir la especificación internacional IEEE802.3 (<http://www.ieee802.org/3>).

"Ethernet" es la tecnología más extendida con diferencia en la construcción de redes de área local (las llamadas "LAN" –Local Area Network–); estas redes permiten el tráfico de datos entre computadores, impresoras, "routers", etc., dentro de una zona de distancias relativamente reducidas, tal como un edificio o un conjunto de edificios adyacentes.

Aunque técnicamente no tenga por qué, en la práctica las redes Ethernet utilizan también el mismo conjunto de protocolos de comunicación que Internet (la llamada pila "TCP/IP").

El shield Ethernet permite, pues, transferir datos entre él mismo (los cuales pueden ser obtenidos de algún sensor, por ejemplo) y cualquier otro dispositivo conectado a su misma red LAN (normalmente, un computador que los recopila y guarda), o viceversa: transferir datos entre un dispositivo conectado a la LAN (normalmente, un computador ejecutando algún tipo de software de control) y él mismo (el cual puede estar conectado a algún actuador controlado remotamente por ese computador). También se puede lograr, gracias al establecimiento del

CAPÍTULO 2: HARDWARE GENUINO

enrutamiento de paquetes adecuado, comunicar nuestro shield Ethernet con cualquier dispositivo conectado a cualquier red del mundo fuera de nuestra LAN privada (incluyendo "Internet"), con lo que sus posibilidades de uso se disparan.

Evidentemente, solo con disponer de un zócalo RJ-45 el shield Ethernet no es capaz "por arte de magia" de comunicarse dentro de una red. Lo que permite que esto ocurra es, tal como ya hemos comentado, la inclusión dentro de este shield de un chip controlador de Ethernet (el W5100) que, de hecho, implementa por hardware toda la pila de protocolos TCP/IP (en concreto los protocolos TCP, UDP, ICMP, Ipv4, ARP, IGMPv2, PPPoE y por supuesto Ethernet). Este chip es pues el verdadero responsable de que el shield Ethernet pueda manejar redes de este tipo.

Que este chip "implemente por hardware la pila TCP/IP" significa en pocas palabras que permite al desarrollador de programas Arduino utilizar de forma muy sencilla la red (mediante la librería de programación "Ethernet" ya mencionada) para poder transmitir o recibir datos sin tener que preocuparse por detalles más técnicos (como el control de errores en la transmisión de datos, la sincronización de las señales y datagramas, etc.) de los cuales ya se encarga directamente este chip.

Entre otras características, conviene saber que el W5100 puede funcionar a velocidades de transmisión de 10 y 100 megabits por segundo, que permite hasta cuatro conexiones simultáneas independientes y que dispone de una memoria interna de 16 kilobytes para almacenar temporalmente datos enviados o recibidos de la red. El W5100 se comunica mediante el protocolo SPI con el ATmega328P de la placa subyacente, por lo que hay que tener en cuenta que, en el caso de que esta sea la placa UNO, los pines digitales números 10, 11, 12 y 13 del shield Ethernet estarán reservados (es decir, no se podrán utilizar para otra cosa); esto implica que, realmente, al usar este shield tenemos 4 entradas/salidas digitales menos.

Este shield también incorpora un zócalo para colocar una tarjeta microSD, la cual se podrá utilizar mediante la librería de programación "SD" (que viene por defecto en el lenguaje Arduino oficial) para guardar diferentes tipos de ficheros y ofrecerlos a través de la red (que recordemos, puede ser nuestra LAN privada o, si queremos, el mundo entero). Al igual que ocurre con el chip W5100, para poder comunicarse con la tarjeta microSD la placa Arduino subyacente utiliza el protocolo SPI, pero para ello esta vez emplea como pin SS el número 4 en vez del 10 (reservado al W5100). Es decir, emplea los pines 4, 11, 12 y 13. En cualquier caso, si no quisieramos utilizar la tarjeta microSD (o bien el chip W5100), hay que tener la precaución de desactivar explícitamente el dispositivo no deseado; esto se haría dentro del código de nuestro programa: para desactivar la tarjeta microSD se debería configurar el pin 4 como salida y para desactivar el chip W5100 se debería hacer lo mismo pero con el pin 10.

EL MUNDO GENUINO-ARDUINO

No está de más indicar que este shield también dispone de su propio botón de "reset", el cual reinicia tanto el chip W5100 como la propia placa Arduino, y de una serie de LEDs informativos interesantes de conocer: "PWR" (indica que la placa y el shield reciben alimentación eléctrica), "LINK" (indica la presencia de una conexión de red, y parpadea cuando el shield recibe o transmite datos), "FULLD" (indica que la conexión de red es "full duplex"), "100M" (indica la presencia de una conexión de red de 100 megabits/s –en vez de una de 10 megabits/s–), "RX" (parpadea cuando el shield recibe datos), "TX" (parpadea cuando el shield envía datos) y "COLL" (parpadea cuando se detectan colisiones de paquetes en la red).

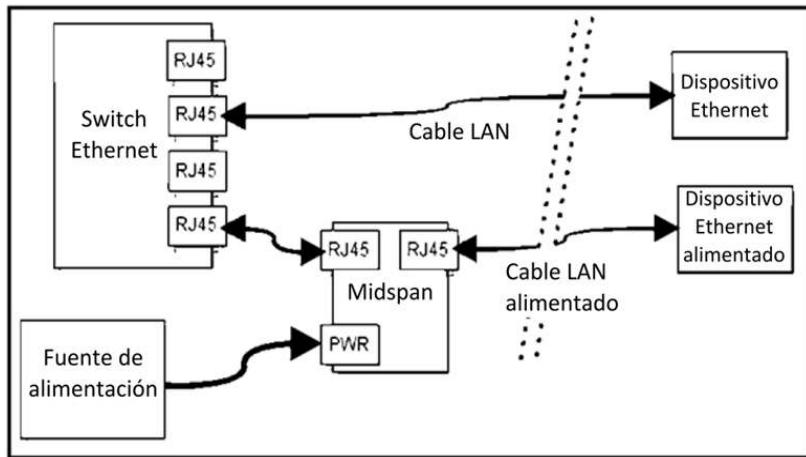
Finalmente, comentaremos que este shield ofrece la posibilidad de acoplársele un módulo PoE. Estudiemos esto.

PoE ("Power Over Ethernet")

Algo que conviene conocer en relación con la placa Arduino Ethernet es que existe la posibilidad de alimentarla eléctricamente a través del propio cable Ethernet, sin usar ni cable USB ni fuente de alimentación externa (adaptador AC/DC, pila, etc.). Es decir, se puede aprovechar la conexión de datos que ofrece el cable de par trenzado típico de una red Ethernet para recibir por allí también el voltaje necesario para el correcto funcionamiento de la placa, sin necesidad, pues, de utilizar ningún otro cable. Esto es muy conveniente en instalaciones que por su tamaño o su ubicación hagan complicado el añadido de varios cables.

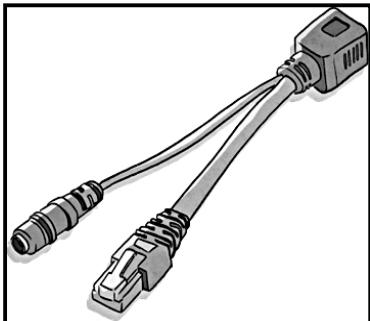
Para ello, no obstante, se han de cumplir diferentes condiciones. La primera es que a través del cable Ethernet viaje la señal adecuadamente transformada. Esto se puede conseguir de dos maneras diferentes:

Utilizando un inyector "midspan". Este es un dispositivo que se enchufa por un lado tanto a la alimentación eléctrica externa (mediante un adaptador AC/DC, generalmente) como a un switch de red estándar (mediante un cable Ethernet), y por otro con el dispositivo a alimentar (mediante otro cable Ethernet). Los datos atravesarán el inyector desde/hacia el switch o desde/hacia el dispositivo final por los cables Ethernet de ambos tramos sin alteraciones, pero además, la alimentación eléctrica recibida por el inyector será transmitida por el cable Ethernet conectado al dispositivo final. Básicamente el esquema descrito sería la siguiente figura:



Utilizando un switch PoE. Este es un switch (un conmutador de red) especial que además de funcionar como un switch Ethernet estándar (intercomunicando entre sí los dispositivos conectados a él), tiene la capacidad de proporcionar alimentación eléctrica a través de sus zócalos RJ-45. Por tanto, los dispositivos que queramos que tanto estén conectados entre sí vía Ethernet como que estén alimentados eléctricamente vía PoE debemos enchufarlos a este tipo de switch, mediante un cable Ethernet estándar cualquiera.

Actualmente existe dos especificaciones PoE estándar: la llamada "802.3af" (del 2003) o, más moderna y compatible con la anterior, "802.3at" (del 2009, también llamada "PoE+"). Ambos documentos definen, tanto para los inyectores midspan como para los switches PoE, qué voltajes, polaridades, disposiciones de pines, etc., han de seguir. No obstante, estos estándares imponen a los dispositivos involucrados un valor de potencia de trabajo de alrededor de 13W –PoE– o 25W –PoE+–, lo que en la práctica implica voltajes máximos de trabajo de más de 40V (e incluso de más de 50V), mucho más del voltaje que los componentes usados en nuestros proyectos pueden manejar (incluso para los reguladores de tensión). Además, el esquema de señalizaciones que ha de implementar el dispositivo a alimentar para informar al inyector sobre la cantidad de tensión que requiere es relativamente complejo. Por ello, muchas veces se suele utilizar un sistema simplificado que utiliza las mismas conexiones que el estándar 802.3 pero que funciona a una tensión más reducida, y sin esquema de señalizaciones.



En otras palabras: para poder utilizar PoE con Arduino no es necesario utilizar un switch o un inyector "midspan" que soporte el estándar 802.3 al completo, los cuales resultan ser además bastante caros. Podemos utilizar simplemente un inyector "midspan" tan elemental como el que forma parte del producto **nº 10759** de Sparkfun (este producto también incluye además un splitter PoE –enseguida veremos qué es esto–). Este inyector (mostrado en la ilustración de la izquierda) consta simplemente por un lado de un conector "jack" de 2,1mm hembra para empalmar con la fuente de alimentación y un conector RJ-45 para enchufar en un switch, y por el otro tiene un zócalo RJ-45 donde se colocará el cable Ethernet que llevará la señal PoE al dispositivo alimentar. Sparkfun lo vende junto con un splitter PoE (enseguida veremos qué es esto).

Otro inyector "midspan" algo más sofisticado pero también muy barato y sencillo de usar es el llamado **4-Channel PoE Midspan Injector** fabricado por Freetronics, el cual ofrece un voltaje de hasta 24V (de sobra para lo que una placa Arduino es capaz de manejar). Concretamente, este inyector consta de un conector de alimentación "jack" de 2,1mm preparado para ser conectado a una fuente de alimentación que proporcione un voltaje de entre 7V y 24V (AC o DC), 4 zócalos RJ-45 pensados para ser conectados a un switch de red estándar y 4 zócalos RJ-45 pensados para ser conectados a distintos dispositivos a alimentar. Esta alimentación se realizará gracias a un voltaje de salida del inyector que será de idéntico valor al de entrada pero siempre DC. Por lo tanto, según el valor de la tensión aplicada al inyector, es posible que se necesite acoplar entre este y el dispositivo un regulador de tensión que la rebaje hasta valores más aceptables para nuestras placas Arduino.

Otra condición que se ha de cumplir para poder utilizar PoE es que el dispositivo alimentado (y conectado a la vez) por el cable Ethernet disponga de la capacidad de procesar correctamente la señal PoE que le llegue por dicho cable. Si ese dispositivo ya de por sí está habilitado para manejar señales PoE, no hay más que hacer: simplemente se le conecta el cable Ethernet y listo. Si ese dispositivo, en cambio, no es capaz de interpretar las señales PoE, tan solo entenderá la comunicación de datos pero no podrá recibir alimentación eléctrica. Para conseguir esto, se puede hacer de varias maneras:

Utilizar un "splitter PoE". Este no es más que un cable que por un lado tiene un zócalo RJ-45 donde se ha de insertar el cable Ethernet que transmite la señal PoE y que por el otro se divide en dos conectores a enchufar al

dispositivo: un conector RJ-45 que transmite los datos y un conector "jack" de 2,1mm, que transmite la corriente continua. El producto nº10759 comentado en los párrafos anteriores incluye un "splitter PoE" como el mostrado en la siguiente figura:



Acoplar una pequeña plaquita extra (también llamado "módulo") al dispositivo en cuestión, que dote a este de la capacidad de procesar la señal PoE convenientemente recibida a través del cable Ethernet. En el caso concreto de la placa Arduino Ethernet, se puede adquirir el módulo PoE **Ag9712-S**, específicamente adaptado a ella, el cual ofrece 9V de salida a la placa con un rango de voltaje de entrada por el cable LAN de entre 36V y 48V. No obstante, este módulo no es fabricado oficialmente por el Arduino Team porque es hardware privativo. Se puede adquirir bien en las webs de los distintos distribuidores listados en el apéndice A, o bien en la web del propio fabricante (http://www.silvertel.com/poe_products.htm).

Arduino WiFi Shield 101

Este shield está pensado para añadir a una placa Arduino (UNO, Mega, Due, Zero...) la capacidad de conectarse inalámbricamente a una red TCP/IP. Para ello incorpora el chip ATWINC1500 de Atmel, el cual incluye una antena integrada y permite conectarse a redes Wi-Fi de tipo 802.11b, 802.11g y 802.11n (hasta 72Mbits/s). Este shield incorpora además otro chip de apoyo al anterior, el ATECC508A (también de Atmel) el cual integra en hardware un par de algoritmos criptográficos de tipo curva elíptica (ECC) –concretamente, el ECDH (para gestionar el intercambio de claves entre los extremos, proceso necesario a fin de poder establecer correctamente una comunicación cifrada entre ellos) y el ECDSA (para firmar un mensaje, o verificar la firma de otro, y así garantizar la autenticidad e integridad del mensaje enviado o recibido, respectivamente)– que permiten comunicaciones confidenciales y seguras entre dos extremos.

Las redes a las que se puede conectar pueden ser abiertas, o bien estar protegidas mediante encriptación de tipo WEP, WPA2-Personal o WPA2-Enterprise. En cualquier caso, solo podrá conectarse a una red si esta emite públicamente su SSID (es decir, si su SSID no está oculto). Para gestionar este shield se debe usar la librería oficial "WiFi101", la cual viene por defecto en el propio lenguaje Arduino.

EL MUNDO GENUINO-ARDUINO

Igual que ocurre con el resto de shields oficiales, una vez conectado este shield sobre una placa Arduino (gracias a la ristra de pines que encaja perfectamente arriba y abajo) para nuestros circuitos utilizaremos a partir de entonces las entradas y salidas ofrecidas por los pines-hembra de este shield. Estas entradas y salidas tienen exactamente la misma disposición y funcionalidad que las de las placas Arduino (UNO, Mega, Due, Zero...). Por tanto, si fuera necesario se podría conectar sin problemas un segundo shield en la parte superior de este shield para seguir sumando funcionalidad.

El procedimiento de cargar nuestros programas en el microcontrolador de la placa que esté acoplada al shield WiFi no varía respecto al realizado normalmente con una placa independiente: tan solo hay que conectarla a nuestro computador mediante el cable USB y utilizar la opción pertinente en el entorno de desarrollo. Tras haber cargado el programa, como siempre, podremos seguir alimentando el conjunto placa más shield vía USB o bien desconectarlo del computador y enchufarlo a una fuente de alimentación externa. El shield requiere 5V para funcionar, pero este voltaje lo aporta la placa subyacente mediante el encaje del pin de alimentación "Vin" de la placa con el correspondiente del shield.

La comunicación entre el chip ATWINC1500 y la placa subyacente se establece vía SPI mediante los pines ICSP más el pin-hembra nº 10 (usado como SS) y también el pin nº 7, por lo que no podremos usarlos como entrada o salida estándar.

Finalmente, indicar que este shield también dispone de su propio botón de "reset" (el cual reinicia tanto el chip ATWINC1500 como la propia placa Arduino) y también de una serie de LEDs informativos: "ON" (iluminado permanentemente si el shield está alimentado), "WIFI" (iluminado permanentemente si hay establecida una conexión a una red), "ERROR" (iluminado solo cuando hay un error en la comunicación) y "NETWORK" (iluminado solo cuando hay datos transmitiéndose o recibiéndose en ese momento).

Arduino GSM Shield

GSM ("Global System for Mobile") es un sistema estándar, libre de regalías, de telefonía móvil digital, comúnmente denominado simplemente "2G" ("segunda generación"). En realidad, a pesar de su nombre, este shield es capaz de conectarse también a redes de tipo GPRS ("General Packet Radio Service"), las cuales aportan varias mejoras sobre las redes GSM tradicionales (sobre todo en relación con la velocidad de transferencia de datos binarios, porque la calidad en la transmisión de voz sigue siendo similar), de ahí que se llamen "2.5G". En concreto, este shield (así como cualquier teléfono móvil compatible con una red GPRS) es capaz, además de

enviar y recibir llamadas de voz y mensajes cortos de texto (SMS) o multimedia (MMS), de conectarse con un computador y enviar y recibir mensajes por correo electrónico o por diferentes sistemas de mensajería instantánea, de navegar por Internet, de acceder con seguridad a la red informática de una compañía (red local/Intranet), etc.

No obstante, actualmente son más populares las redes de tipo UMTS (3G), HDPA (3.5G) y LTE (4G), mucho más rápidas y fiables que las de tipo GPRS pero, desgraciadamente, este shield no es compatible con este tipo de redes.

Arduino Motor Shield

Este shield incorpora el chip L298P del fabricante STMicroelectronics (la "P" final simplemente indica el tipo de encapsulado que tiene, ya que para el mismo chip L298 existen otras formas y tamaños, identificados con otras letras). Este chip está diseñado para controlar componentes que contienen inductores (es decir, "bobinas") en su estructura interna, tales como relés, solenoides, motores de corriente continua –DC–o motores paso a paso –"steppers"–, entre otros. En concreto, gracias al chip L298P, el shield Arduino Motor nos permite dominar la velocidad y sentido de giro de hasta dos motores DC de forma independiente o bien estas dos magnitudes de un motor paso a paso. También podremos realizar diferentes medidas sobre las capacidades de los motores conectados.

Como el voltaje necesario para el correcto funcionamiento de los motores suele sobrepasar el aportado por un simple cable USB, este shield siempre necesitará ser alimentado por una fuente externa. Esta puede ser bien un adaptador AC/DC que aporte entre 7 y 12V DC de salida y conectado al jack de 5,5/2,1mm de la placa Arduino subyacente, o bien una pila (preferiblemente de 9V) conectada mediante cables directamente a los bornes de los tornillos etiquetados como "Vin" y "Gnd" del propio shield. En ambos casos (usando el adaptador AC/DC o la pila), estaremos alimentando a la vez tanto al shield como a la placa.

Si los motores utilizados requirieran más de 9V, entonces sería necesario separar las líneas de alimentación de placa y shield para que cada una se pudiera alimentar por separado: esto se hace cortando la soldadura existente entre los extremos del metal etiquetado como "Vin connect" que aparece en el dorso del shield. De esta forma, el shield se podría continuar alimentando mediante la fuente conectada a sus bornes de tornillo "Vin" y "Gnd", y la placa por su lado se alimentaría de la forma habitual (con el cable USB o bien con la fuente externa que estuviera conectada al jack de 5,5/2,1mm). De todas formas, hay que tener en cuenta que, en cualquier caso, el máximo voltaje que admiten los bornes de tornillo es de 18V.

EL MUNDO GENUINO-ARDUINO

Este shield tiene dos canales separados (etiquetados como "A" y "B"). Cada uno de estos canales ofrece dos bornes de tornillo (etiquetados como "+" y "-" en los dos) para conectar allí los motores. Cada canal por separado puede manejar un motor DC independiente, pero también se pueden combinar, tal como se ha mencionado, para manejar entre los dos un único motor paso a paso. Si estamos en el primer caso (es decir, si queremos manejar uno o dos motores DC), deberemos emplear determinados pines-hembra del shield para controlar y monitorizar el motor DC conectado a cada canal. Esto implica que esos pines-hembra no los podremos usar para otra cosa, por lo que dispondremos de menos entradas/salidas de propósito general. En concreto, la función de estos pines-hembra especializados es la siguiente:

Función	Pines Canal A	Pines Canal B
Sentido de movimiento	D12	D13
Velocidad (PWM)	D3	D11
Freno	D9	D8
Detección de corriente	A0	A1

A partir de la tabla anterior se puede deducir cómo es el funcionamiento del shield: después de conectar el par de cables que proporcione cada motor DC a los terminales "+" y "-" de su respectivo canal (A o B), para controlar su sentido de movimiento deberemos enviar una señal de valor HIGH o LOW (según el sentido de giro deseado) a los pines de dirección correspondientes. Para controlar la velocidad de giro deberemos variar los valores de la señal PWM en los pines correspondientes. Finalmente, los pines de freno si tienen el valor HIGH paran en seco los motores DC asociados (en vez de dejarlos desacelerar paulatinamente como ocurriría si se les corta la alimentación). Por último, se podría conocer el valor de la intensidad de corriente que pasa a través del motor DC leyendo el valor de la señal existente en los pines correspondientes de la tabla anterior, ya que la señal recibida es proporcional a dicha intensidad (cada canal puede recibir una corriente de 2 amperios como máximo, la cual se corresponde a una señal máxima posible de 3,3V).

NOTA: Si no necesitáramos la funcionalidad de freno o la de detección de corriente y necesitáramos más pines para nuestro proyecto, se podrían desactivar estas funcionalidades cortando la soldadura existente entre los extremos de las respectivas zonas de metal etiquetadas convenientemente en el dorso del shield.

Este shield dispone además de una serie de zócalos de tipo "Tinkerkit" (concretamente 2 zócalos blancos de 3 pines para poder enchufar sendos dispositivos funcionando como entradas analógicas que ofrezcan una clavija compatible –los cuales serán conectados internamente, además de a alimentación y tierra, a los pines

A2 y A3–; 2 conectores naranja de 3 pines para poder enchufar sendos dispositivos compatibles funcionando como salidas analógicas –conectadas internamente a las salidas PWM de los pines D5 y D6– y 2 conectores blancos de 4 pines, uno para entrada I²C/TWI y otro para salida I²C/TWI). Desgraciadamente, este tipo de conectores está actualmente en desuso.

Arduino Proto Shield

Este shield permite de forma fácil diseñar e implementar nuestros circuitos personales. Básicamente lo que hace es ofrecer un área de trabajo donde se pueden soldar (usando tanto la técnica del montaje superficial –SMT–, como la técnica de los agujeros pasantes –THT–) los diferentes componentes electrónicos que necesitemos para montar nuestro proyecto. De esta forma, podemos tener de una forma muy compacta todo un circuito completo formado por placa, shield y componentes conectados, todo en uno.

Recordemos que en este libro no se realizará ningún proyecto donde sea necesario soldar componentes. Si bien es cierto realizar soldaduras domésticas (sobre todo de tipo THT) no es excesivamente complicado y pueden conseguirse resultados muy buenos con material y herramientas relativamente baratas sin problemas, hay que reconocer que se requiere cierta experiencia y pericia en este tema. Por eso, y ya que este texto no deja de ser una introducción al mundo de la electrónica, no se ha considerado oportuno profundizar en este asunto, que tal vez pertenezca a un peldaño inmediatamente superior de conocimientos.

Si no se desea soldar ningún componente (como es nuestro caso), este shield también nos puede ser igualmente útil porque permite colocar de forma permanente (en un área especialmente reservada para ello) una pequeña breadboard de 1,8x1,4 pulgadas (adquirida aparte). Así podremos enchufar rápidamente todos los elementos del circuito y comprobar que este funcione correctamente de una forma muy compacta y segura.

Este shield recibe la alimentación de los pines estándares 5V y GND de la placa Arduino y la reparte a lo largo de dos filas de once agujeros etiquetadas en el shield claramente como "5V" y "GND". A partir de estas dos filas se puede alimentar fácilmente cualquier componente (incluso zócalos DIP) que se suelde a la placa.

Otras características de este shield son la presencia de un botón de reinicio y de un conector ICSP (comunicado al conector ICSP base) para no tener que usar los ubicados en la placa subyacente, y la disposición de todos los pines de E/S idéntica a la de los de la placa Arduino UNO, para poder trabajar con ellos de la forma habitual.

¿QUÉ SHIELDS NO OFICIALES EXISTEN?

Existen gran cantidad de shields diseñados y construidos por la comunidad que ofrecen soluciones a necesidades específicas que los shields oficiales no presentan (o presentan de otra forma).

Proto Shields

Un ejemplo de shield que nos puede ser útil en varios proyectos diferentes es el llamado genéricamente "proto shields". Básicamente, lo que ofrecen este tipo de shields es un área de prototipado para implementar los circuitos de una forma mucho más compacta que si lo hiciéramos mediante una breadboard por separado (además de algún pulsador, LED o potenciómetros extra, según el modelo). Un ejemplo de este tipo de shield lo acabamos de ver: el Arduino Proto Shield oficial, pero otros similares fabricados por terceros son, por ejemplo, el "**Prototyping Shield**" de Cytron Technologies, el "**Electronic Brick Shield**" de MakerStudio, el "**Assembled Protoshield**" de LinksSprite (aunque todos ellos, carecen, no obstante, de zona SMT). A destacar en este sentido especialmente el producto **DFR0019** de DFRobot, shield que, en vez de ofrecer una zona THT, incorpora directamente una mini-breadboard, lo que permite realizar las conexiones necesarias de nuestros circuitos sin tener que soldar.

Desgraciadamente, muchos "proto shields" se distribuyen en forma de kit (es decir, con sus piezas sin ensamblar), por lo que es necesario, una vez adquiridos, soldar todas sus partes para obtener un shield funcional. Ejemplos de estos últimos son el producto **MSMS01** de Makershed, el producto **nº 2077** de Adafruit, el producto **nº 103060000** de Seeedstudio, el producto **nº 7914** de Sparkfun, el **IM120628019** de IteadStudio o el **ARD-0119-NKC** de NKC Electronics. Freetronics incluso tan solo distribuye un PCB del shield, sin componentes adicionales, con el nombre de "**Protoshield Basic**".

Una alternativa a los "proto shields" anteriores son los llamados genéricamente "screw shields", los cuales además de ofrecer igualmente una superficie diseñada para soldar los componentes (generalmente de tipo THT) de nuestros circuitos y ubicarlos así de forma compacta, sustituyen los pines-hembra laterales por terminales de tornillo de 3,5mm, ideales para fijar de una forma mucho más estable los cables a conectar a los pines de entrada/salida/alimentación mediante destornilladores. Ejemplos son el "**Terminal Shield for Arduino**" de Freetronics, el "**Screw Terminal Shield**" de Cytron, el producto **DFR0131** de DFRobot, el producto **nº 9729** de Sparkfun (viene en forma kit), el producto **nº 196** de Adafruit (también en forma de kit), el **IM120417013** de IteadStudio o el "**Power ScrewShield**" de Snootlab.

NOTA: Si lo que quisiéramos fuera disfrutar de los terminales de tornillo pero no de la superficie de prototipado, podríamos entonces acoplar sobre cada uno de las dos ristras de pines-hembra laterales de una placa Arduino sendas "alas", como son por ejemplo el producto **DFR0060** (o la versión mejorada **DFR0171**) de DFRobot, el "**Screw Shield**" de TinySine o el producto homónimo de Solid Depot, entre otros.

Power Shields

Otro tipo de shields diferentes de los anteriores pero que conviene conocer porque también nos pueden ser de gran ayuda en múltiples proyectos son los que permiten situar sobre sí mismos algún tipo de batería de tal forma que se consiga una estructura sólida y compacta de fuente de alimentación más placa Arduino, haciendo al conjunto energéticamente independiente de nuestro computador. Un ejemplo de este tipo de shield es el "**Lithium BackPack**" de LiquidWare, el cual admite la colocación (mediante conector JST) de una batería Li-Ion de varios tipos compatibles (2200mAh, 1000mAh o 660mAh), todos ellos capaces de alimentar nuestra placa Arduino; también tiene la opción de actuar como cargador de la batería, bien a través de la alimentación recibida por la placa Arduino, o bien directamente del exterior mediante un conector USB mini-B ya incluido.

Otros shield similar al anterior es el producto **nº 13158** de Sparkfun. Este shield también es capaz también de realizar dos funciones: se puede utilizar como cargador de una batería (en este caso, de tipo LiPo de 3,7V y preferiblemente 500mAh) y además como fuente de alimentación de la placa Arduino. La circuitería interna del shield se encarga de proteger la batería de cargas, descargas o corrientes excesivas, y de convertir los 3,7V ofrecidos por ella en los 5V necesarios para hacer funcionar Arduino, por lo que tan solo nos deberemos preocupar de conectar la batería LiPo al zócalo de tipo JST incorporado en el shield y ya está. Si se desea recargar la batería, lo podemos hacer a través de la alimentación recibida por la propia placa Arduino o bien directamente del exterior mediante un conector USB mini-B que viene incluido. Este shield permite además monitorizar el estado de carga de la batería mediante un código Arduino específico, descargable de la página de este producto.

Otro shield similar es el producto **nº 2078** de Adafruit, el cual permite conectar al zócalo JST que lleva incorporado una batería de tipo Li-Ion (o bien Li-Po) de 4,2V/3,7V con capacidades de entre 1200mAh y 2000mAh para alimentar la placa Arduino subyacente aportando como mínimo 500mA (y como máximo 1A). También incorpora un conector USB micro-B a través del cual puede recibir la alimentación necesaria para recargar la batería acoplada (al desconectar el cable USB automáticamente la batería empezará a funcionar). También incorpora un

EL MUNDO GENUINO-ARDUINO

interruptor para abrir o cerrar la corriente hacia la placa. Un shield similar es el llamado "**Energy Shield**" de Seeedstudio, el cual, como diferencia más importante, permite recargar la batería mediante tres posibles métodos: a través de una fuente de alimentación conectada al zócalo USB o bien, 5,5/2,1mm de la placa Arduino subyacente, o a través de un eventual panel solar conectado a otro zócalo JST presente en el propio shield (y activado mediante un conmutador). Este shield también ofrece un conector USB de tipo A para cargar a su vez otros dispositivos, como teléfonos móviles, por ejemplo.

Otro shield interesante, aunque esta vez algo diferente, es el producto **IM121115001** de IteadStudio, el cual simplemente aloja un soporte de dos pilas AAA y un conversor DC-DC para poder alimentar la placa Arduino subyacente con una tensión de 5V y una corriente de 350mA.

De todas formas, es imposible nombrar aquí todos los shields que puedan ser destacables por alguna razón: son simplemente demasiados y muy diferentes entre sí. Muchos de ellos los iremos conociendo a medida que necesitemos utilizarlos en los diferentes proyectos existentes a lo largo del libro, pero aun así quedarán en el tintero muchos que nos puede ser útil conocer. Recomiendo, pues, consultar el extenso catálogo de los fabricantes y diseñadores de shields más importantes, como Sparkfun, Adafruit, Seeedstudio, Iteadstudio, Freetronics, Olimex o DFRobot, entre otros listados en el apéndice A.

¿QUÉ PLACAS ARDUINO NO OFICIALES EXISTEN?

Que los diseños de las placas Arduino tengan una licencia libre implica, entre otras cosas, que cualquier persona o empresa puede realizar, a partir de ellos, sus propios diseños derivados (y fabricar sus propias placas siguiendo estos nuevos diseños) sin necesidad de pedir permiso al Arduino Team; en otras palabras, todo el mundo es libre para estudiar, modificar y mejorar los diseños originales de Arduino y para fabricar y vender placas basadas en estos. El único requisito legal indispensable es señalar claramente que la placa derivada no es oficial de Arduino.

Esta facilidad para manufacturar y comercializar placas "compatibles" provoca, tal como ya hemos comentado, que exista una gran proliferación de placas más o menos similares a las oficiales Arduino (sobre todo al modelo UNO, el más popular), aunque solo algunas pocas contengan mejoras interesantes: en la práctica, la gran mayoría de placas derivadas no difieren apenas del diseño original, tal vez el cambio más habitual es la sustitución del chip ATmega16U2 por otro chip más

especializado (como el propio FT232RL, tal como hace, por ejemplo, la placa "Red Board" de Sparkfun, u otros chips aún más exóticos como el CH340G), así que, en realidad, tan solo son atractivas por tener un precio de venta menor. En esos casos, no obstante, el lector debería ser consciente de que adquiriendo un "clon" barato de Arduino no colabora con el mantenimiento del proyecto Arduino oficial.

Para que el lector pueda valorar si algún modelo concreto de placa no oficial puede ser útil en sus proyectos, a continuación mostraremos una lista de los fabricantes de placas compatibles más representativos. Por "placa compatible" se entiende una placa no oficial que sea capaz de ejecutar programas escritos con el lenguaje Arduino oficial y compilados/cargados con el IDE oficial. No obstante, en el siguiente listado solo se han incluido aquellas placas derivadas que aportan alguna novedad interesante (en relación con su posible versatilidad, funcionalidad o especialización) más allá de ser un simple "clon" barato del correspondiente modelo oficial.

Freetronics (<http://www.freetronics.com>): Entre otras placas, ofrece una propia llamada "Eleven", que es una modificación de la UNO, donde (entre otras ventajas) sustituye el conector USB tipo B por uno de tipo micro-B, añade una zona de prototipado en la propia placa aprovechando el espacio disponible y sustituye el reloj cerámico del ATmega328P por uno de cristal. Otra placa suya es la llamada "USBxDroid", que es una modificación de la UNO a la que se le ha añadido un zócalo USB de tipo A extra y el chip MAX3421E para convertirla en un "host USB" y así poder desarrollar proyectos que hagan uso de la librería USBHost oficial (hablaremos de ella en próximos capítulos), además de un zócalo microSD y una pequeña zona de prototipado. Otra placa suya es la llamada "Etherten", que, además de sustituir el zócalo USB tipo B por otro micro-B y de incorporar un zócalo microSD, une la funcionalidad de la placa UNO con el del shield Arduino Ethernet ya que incorpora un conector RJ-45, el mismo chip Wiznet W5100 y soporte para PoE todo en uno. Otra placa suya es la llamada "EtherMega", compatible con la placa Mega pero un conector RJ-45 añadido (junto con el chip Wiznet W5100 y capacidad para PoE), un zócalo microSD, una zona de prototipado y un conector USB micro-B en vez del B estándar. De forma análoga también distribuyen la placa "EtherDue", similar a la Due oficial pero con un zócalo microSD y un conector RJ-45 y el chip Wiznet W5100 añadidos. Otra placa suya es la llamada "LeoStick", compatible con la placa Arduino Micro pero con el tamaño de un lápiz USB (y como tal, conectable directamente a un zócalo USB de cualquier computador).

EL MUNDO GENUINO-ARDUINO

Seeedstudio (<http://www.seeedstudio.com>): fabrica varias placas llamadas "Seeeduino", cada una compatible con un modelo diferente de placa oficial: UNO, Mega, etc. En el caso de la Seeeduino UNO, las modificaciones más importantes son las siguientes: es capaz de trabajar a 5V o a 3,3V (para ser compatible con los sensores y dispositivos más modernos) dependiendo de la posición de un interruptor específico ubicado en la propia placa, duplica los pines de entrada/salida digitales y los de entrada analógicos, incorpora un conector USB de tipo micro-B, ofrece pines para comunicarse directamente con el microcontrolador ATmega16U2 y un conector extra de tipo I²C y otro de tipo serie (en formato Grove). También podemos adquirir su Seeeduino Mega, cuyas características más relevantes, además de también disponer de un selector para determinar la tensión de trabajo (5V o 3,3V), es su forma reducida (compatible con los shields diseñados para placas del tamaño de la UNO), la presencia, en vez de un zócalo "jack" de 5,5/2,1mm, de un zócalo JST de 2 pines para recibir la alimentación externa y la duplicación de los pines de entrada/salida digitales y los de entrada analógicos. Conviene decir que todos estos modelos de placas (y los demás que fabrica Seeedstudio) disponen de una completísima y muy recomendable página wiki propia de la empresa en la cual se explican todas las características de dichas placas y los aspectos necesarios para aprender paso a paso a sacarles el máximo provecho.

Iteadstudio (<http://www.iteadstudio.com>): fabrica varias placas llamadas "Iteaduino", cada una compatible con un modelo diferente de placa oficial: UNO, Mega, Due, etc. En el caso de la Iteaduino UNO, las modificaciones más importantes son las siguientes: es capaz de trabajar a 5V o a 3,3V (para ser compatible con los sensores y dispositivos más modernos) dependiendo de la posición de un interruptor específico ubicado en la propia placa, reproduce en múltiples sitios de la placa series de tomas de tierra, alimentación, pines de entrada/salida digitales y pines de entrada analógicos, de manera que sea mucho más cómodo conectar todo tipo de sensores y actuadores sin necesidad de usar una breadboard, e incorpora un conector USB de tipo micro-B. Otra placa que fabrica esta empresa compatible con Arduino UNO (programable mediante un adaptador FTDI USB<->Serie) es la llamada "IBoard", la cual dispone, además de un zócalo para una tarjeta microSD, de un conector RJ-45 y el módulo Wiznet W5100 para ofrecer conectividad Ethernet compatible con PoE. En el caso de la Iteaduino Mega, las modificaciones más importantes respecto a la placa oficial son: la presencia de un selector para determinar la tensión de trabajo (5V o 3,3V), su forma reducida (compatible con los shields diseñados para placas del tamaño de la

CAPÍTULO 2: HARDWARE GENUINO

UNO) y la ampliación del número de pines de entrada/salida. También ofrece la placa llamada "WBoard", basada en el modelo Mega pero con un chip WiFi integrado (el ESP8266), y más variantes aún, listadas todas ellas en la página <http://www.itead.cc/development-platform/arduino/arduino-compatible-mainboard.html?dir=asc&order=price#/page/1>.

DFRobot (<http://www.dfrobot.com>): además de fabricar la placa llamada "DFRduino UNO", clon de la placa UNO oficial, también fabrica otras placas compatibles con el modelo UNO como es la placa "Romeo", especialmente pensada para aplicaciones de robótica (con controladores específicos para motores, más pines de entrada y salida, zócalos para módulos inalámbricos, etc.) o la placa "Xboard v2", que incorpora un zócalo RJ-45 y el chip W5100 para ofrecer conectividad Ethernet, así como un zócalo de tipo XBee para ofrecer conectividad inalámbrica.

Olimex (<https://www.olimex.com/Products/Duino/AVR>): fabrica la placa "Olimexino", que es una placa clon del Arduino DueMilanove (es decir, el modelo previo a UNO, ya descatalogado pero compatible a nivel de código) con una serie de mejoras muy interesantes, como la sustitución del conector USB por uno de tipo mini-B, la inclusión de un conmutador para cambiar el voltaje de funcionamiento de la placa entre 5V y 3,3V, la ampliación del rango de voltaje de entrada y de temperatura (haciéndolo más resistente a entornos industriales), la posibilidad de trabajar a varias frecuencias diferentes de 16MHz (gracias a la fácil sustitución de su oscilador de cristal), la inclusión de un conector JST-PH que permite alimentar la placa mediante una batería LiPo y de un circuito cargador que permite recargar esa batería si la placa está siendo alimentada en ese momento vía USB o vía zócalo "jack", el rediseño y reubicación de los elementos de la placa, etc. También ofrecen la placa "Olimexino-32U4", similar a la anterior pero incorporando el microcontrolador ATmega32U4 o la placa "T32U4 Board" (también con el microcontrolador ATmega32U4 pero con forma de "T" y especialmente diseñada para ser conectada sobre una breadboard). Por otro lado, **Pololu** (<https://www.pololu.com>) distribuye una placa parecida a la "Olimexino-32U4" llamada "A-Star 32U4 Prime", la cual viene en dos versiones diferenciadas por el rango de tensión de trabajo admitido (ambos superiores, en todo caso, al de las placas oficiales: la versión "LV" puede funcionar a una tensión entre 3V y 11V y la segunda a una entre 5V y 33V); ambas versiones además pueden venir, opcionalmente, con un zócalo microSD incorporado y/o una pantalla LCD.

EL MUNDO GENUINO-ARDUINO

BQ (<http://www.bq.com>): fabrica la placa "Zum-Core", clon del modelo UNO que reúne tres características interesantes: incorpora un interruptor para cortar la alimentación a la placa fácilmente, aporta un zócalo USB micro-B (en vez de tipo B) y reproduce en múltiples sitios de la placa tomas de tierra, alimentación, pines de entrada/salida digitales y pines de entrada analógica para que sea mucho más cómodo conectar todo tipo de sensores y actuadores sin necesidad de usar una breadboard. Además, incorpora la capacidad de comunicarse vía Bluetooth, incluso para ser programada.

Yourduino (<http://www.yourduino.com>): Además de fabricar varias placas compatibles con diferentes modelos de placa oficial (UNO, Mega, Nano, etc.), también fabrica la llamada "YourduinoRobo1", la cual reproduce en múltiples sitios de la placa tomas de tierra, alimentación, pines PWM y entradas analógicas, de manera que sea mucho más cómodo conectar todo tipo de sensores y actuadores sin necesidad de usar una breadboard. Una placa similar es la "Dev Board" de **ICStation** (<http://www.icstation.com>), la cual, además, aporta un zócalo USB micro-B en vez de tipo B. Por otro lado, otro modelo compatible con UNO que reproduce en múltiples sitios de la placa tomas de tierra, alimentación, pines de entrada/salida digitales, y pines de entrada analógica es el producto 20-011-937 de **Sain Smart** (<http://www.sainsmart.com>).

Ruggeduino (<http://ruggedcircuits.com>): fabrica las placas "Ruggeduino" (clones de los modelos UNO y Mega), las cuales implementan en su circuitería hardware multitud de protecciones eléctricas y de temperatura para evitar posibles cortocircuitos y otras causas de daño permanente, generalmente provocadas por el conexionado incorrecto de los pines de entrada y salida.

ModernDevice (<http://shop.moderndevice.com>): fabrica, entre otras, la placa "RBBB Pro", clon de la UNO cuya característica más relevante es su reducidísimo tamaño, lo que la hace ideal para ser utilizada en textiles, sistemas empotados o cualquier otro proyecto de dimensiones reducidas. En su dorso dispone de pines-macho diseñados para ser encajados en una breadboard, lugar donde podremos colocar el resto de componentes de nuestro circuito y conectarlos cada uno a los pines-macho pertinentes, que se corresponden con las patillas de alimentación, tierra, GPIO, etc. del microcontrolador. Esta placa se programa mediante un adaptador USB-Serie (ModernDevice fabrica uno llamado "BUB"). Por otro lado, un caso más extremo de placa compatible con Arduino UNO de reducido tamaño

(programable también mediante un adaptador USB<->Serie es el "ArdTweeny" de **Solarbotics** (<http://www.solarbotics.com>), cuyas dimensiones apenas superan las del encapsulado DIP del propio microcontrolador; de hecho, es tan reducido que carece de circuito regulado de alimentación: por eso, si se desea, se le puede acoplar (por debajo) una placa suplementaria –también muy pequeña– llamada "ArdTweeny Backpack" la cual incorpora un zócalo "jack" de 5,5/2,1mm y también, para mayor comodidad, pines-hembra asociados a cada uno de los pines-macho del "ArdTweeny". Siguiendo con la misma filosofía de placas compatibles con el modelo UNO pero de tamaño hiperreducido, tenemos la "Sippino" de **Spikenzielabs** (<http://spikenzielabs.com>), la cual tampoco incluye circuito de alimentación regulado (por tanto, se ha de alimentar mediante el pin "5V" o a través del adaptador USB<->Serie) pero como novedad aporta la posibilidad de soldar los pines-macho de forma horizontal, vertical u optar por soldar pines-hembra en su lugar. Spikenzielabs, por su parte, fabrica otras placas interesantes compatibles con UNO, como la llamada "Prototino", la cual contiene sus componentes reubicados de manera que dispone de un gran zona de prototipado para conectar los elementos de nuestros circuitos simples directamente allí sin necesidad de usar una breadboard.

Adafruit (<http://www.adafruit.com>): además de fabricar la placa llamada "Metro", clon de la placa UNO oficial (excepto en que lleva incorporado un interruptor para cortar la alimentación de la placa), distribuye una placa llamada "Boarduino", una versión equivalente a la placa Arduino DueMilanove pero también con un tamaño muy reducido y compacto, especialmente pensada para ser encajada sobre una breadboard. Existen dos modelos: uno que no tiene zócalo de 5,5/2,1mm de alimentación (solo tiene el zócalo USB para poder recibir la energía eléctrica) y otro al revés: que solo tiene el zócalo de 5,5/2,1mm y por tanto, para programarlo se necesita usar un adaptador FTDI USB<->Serie.

JeeLabs (<http://jeelabs.org>): fabrica la placa "JeeNode", que vuelve a ser una placa compatible con el modelo UNO pero otra vez con un tamaño mucho menor. Además, funciona a 3,3V y ofrece conexión inalámbrica de radiofrecuencia a 868 MHz o 915 MHz y bajo consumo. Aunque no dispone de conexión USB, ofrece el chip FTDI FT232RL para poder ser programada mediante el adaptador o cable USB<->Serie correspondiente. Existen diferentes variantes oficiales de "Jeenode", entre las que se encuentra una que contiene los componentes encapsulados en SMD y otra que sí incorpora conexión USB. JeeLabs también ofrece un conjunto muy extenso de módulos

EL MUNDO GENUINO-ARDUINO

de expansión (sensores y actuadores) para aumentar las capacidades de cualquier JeeNode.

Sparkfun (<http://www.sparkfun.com>): además de fabricar la placa llamada "RedBoard", clon de la placa UNO oficial, también fabrica las placas "SAMD21 Dev Breakout" y "SAMD21 Mini Breakout", las cuales –aunque de forma y tamaños diferentes– incorporan ambas el mismo microcontrolador que el de la placa Arduino Zero, el SAM-D21; sin embargo, la circuitería acompañante no es exactamente la misma, por lo que no las podemos considerar clónicas. Se mencionan aquí porque son una alternativa con similares capacidades que la placa oficial pero más económica.

En la lista anterior no se han mencionado la multitud de placas que, pese a no incorporar uno de los modelos de microcontrolador existentes en el ecosistema Arduino, admiten ser programadas igualmente mediante el IDE oficial gracias a la gran flexibilidad y versatilidad de este (eso sí, tras haberlo configurado previamente de forma conveniente siguiendo determinados pasos, particulares a cada caso). Si el lector desea conocer algunas de estas placas, puede empezar por el listado (no exhaustivo) disponible en <https://github.com/arduino/Arduino/wiki/Unofficial-list-of-3rd-party-boards-support-urls>.

3

SOFTWARE ARDUINO

¿QUÉ ES UN IDE?

Un programa es un conjunto concreto de instrucciones, ordenadas y agrupadas de forma adecuada y sin ambigüedades que pretende obtener un resultado determinado. Cuando decimos que un microcontrolador es "programable", estamos diciendo que permite grabar en su memoria de forma permanente (hasta que regrabemos de nuevo si es necesario) el programa que deseemos que dicho microcontrolador execute. Si no introducimos ningún programa en la memoria del microcontrolador, este no sabrá qué hacer.

Las siglas IDE vienen de Integrated Development Environment, lo que traducido a nuestro idioma significa Entorno de Desarrollo Integrado. Esto es simplemente una forma de llamar al conjunto de herramientas software que permite a los programadores poder desarrollar (es decir, básicamente escribir y probar) sus propios programas con comodidad. En el caso de Arduino, necesitamos un IDE que nos permita escribir y editar nuestro programa (también llamado "sketch" en el mundo de Arduino), que nos permita comprobar que no hayamos cometido ningún error y que además nos permita, cuando ya estemos seguros de que el sketch es correcto, grabarlo en la memoria del microcontrolador de la placa Arduino para que este se convierta a partir de entonces en el ejecutor autónomo de dicho programa.

EL MUNDO GENUINO-ARDUINO

Para poder empezar a desarrollar nuestros propios sketches (o probar alguno que tengamos a mano) deberemos instalar en nuestro computador el IDE que nos proporciona el proyecto Arduino. Para ello, seguiremos alguno de los pasos mostrados a continuación, según cada caso particular.

INSTALACIÓN DEL IDE ARDUINO

Cualquier sistema Linux

Para instalar el IDE de Arduino en un sistema Linux cualquiera (sea cual sea la distribución concreta que estemos utilizando: Ubuntu, Arch, Fedora, Debian, etc.), primero debemos ir a <http://arduino.cc/en/Main/Software>, la página de descargas oficial de Arduino. Allí encontraremos dos enlaces diferentes apuntando a sendos archivos, ambos comprimidos en formato "tar.xz": uno de ellos corresponde a la versión del IDE para sistemas de 32 bits y el otro a la versión para sistemas 64 bits. Es necesario, por tanto, conocer primero qué tipo de sistema Linux tenemos instalado (32 bits o 64 bits) y descargarnos entonces el paquete "tar.xz" correspondiente.

Para saber de manera rápida si nuestro sistema Linux es de 32 bits o de 64 bits podemos abrir un terminal y escribir el siguiente comando: `getconf LONG_BIT`. La salida que obtengamos como resultado de ejecutar el comando anterior será "32" si nuestro sistema es de 32 bits y "64" si nuestro sistema es de 64 bits. Una vez sabido esto, ya podremos elegir el paquete "tar.xz" adecuado.

Una vez descargado este paquete "tar.xz", debemos seleccionarlo (haciendo clic con el botón izquierdo del ratón sobre él) y, seguidamente, debemos hacer clic con el botón derecho para que aparezca un menú contextual, en el cual deberemos elegir la opción de "descomprimir" (a veces también listada como "extraer"). El resultado de ello será la aparición, allí donde estemos, de una carpeta (nombrada igual que el paquete "tar.xz") que contendrá los ficheros que forman el IDE. Al entrar dentro de esta carpeta no usaremos ningún instalador (vamos a ignorar, por tanto, el archivo llamado "install.sh") porque nuestro IDE ya está listo para utilizarse desde este mismo momento: tan solo deberemos hacer doble clic sobre un archivo ejecutable (en realidad, un shell script con permisos de ejecución) llamado "arduino" y, después de confirmar la opción "Ejecutar" que aparecerá en el cuadro emergente, veremos el IDE finalmente ante nosotros.

Evidentemente, es posible mover y guardar la carpeta resultado de la descompresión en el lugar que deseemos de nuestro disco duro, siempre y cuando no alteremos nada de su interior. Adicionalmente, también podríamos crear –usando la opción adecuada del menú contextual– un acceso directo al ejecutable "arduino" allí donde lo consideremos oportuno (por ejemplo, en el escritorio).

El IDE de Arduino está desarrollado con el lenguaje Java (tal como lo está el entorno Processing, del cual proviene). Antiguamente esto obligaba al usuario de Linux a tener instalado previamente en su sistema un paquete llamado "Java Runtime Environment" –o abreviando "JRE"– (consistente en un conjunto de utilidades y librerías que permiten precisamente la ejecución de aplicaciones escritas en lenguaje Java) para conseguir un correcto funcionamiento del IDE Arduino. Este paquete (cuya página web oficial es <http://openjdk.java.net>) debía ser descargado e instalado aparte utilizando el "centro de software"/"gestor de paquetes" propio de cada distribución, lo que hacía el proceso poco homogéneo y propenso a errores. Afortunadamente, el IDE actual de Arduino ya incorpora dentro de sí el JRE de forma que hoy en día no es necesario descargarlo desde ningún otro lugar, haciendo, por tanto, que la ejecución del IDE sea mucho más directa y sencilla.

Possible problema: la configuración del gestor de ficheros Nautilus

En el caso de que nuestro sistema Linux utilice por defecto el programa Nautilus como gestor de ficheros (ejemplos de ello son las distribuciones estándar de Ubuntu o Fedora), es posible que al hacer doble clic sobre el fichero "arduino" lo que aparezca sea, en vez del IDE, un editor de texto mostrando el código interno del IDE (algo que no queremos para nada). Para cambiar este comportamiento, deberemos abrir una carpeta cualquiera para así poder acceder a un menú de la barra superior llamado "Edición->Preferencias" (en Ubuntu) o "Fichero->Preferencias" (en Fedora). El cuadro mostrado tras ello contiene la configuración del Nautilus: allí debemos ir a la pestaña "Comportamiento" y seleccionar la opción "Preguntar cada vez" dentro de la sección "Ficheros de texto ejecutables". Así conseguiremos que se nos pregunte si queremos ejecutar el IDE cada vez que hagamos clic sobre el fichero "arduino". Otra manera diferente de conseguir lo mismo es ejecutando el siguiente comando:

```
gsettings set org.gnome.nautilus.preferences executable-text-activation ask
```

Possible problema: los permisos de usuario

Para poder ejecutar el IDE de Arduino sin problemas debemos asegurarnos de que nuestro usuario del sistema tenga los permisos necesarios para que, al utilizar el IDE, este pueda comunicarse a través del cable USB con la placa; si nuestro usuario no tiene esos permisos, el IDE no podrá cargar programas en el microcontrolador. En la práctica, esto se traduce en añadir nuestro usuario al grupo de usuarios "dialout". Este grupo de usuarios (ya predefinido en la gran mayoría de sistemas Linux) tiene asignados los permisos correctos para poder establecer una comunicación serie con las placas Arduino que se conecten al computador. Para añadir, por tanto, nuestro usuario a este grupo, tan solo hay que abrir un intérprete de comandos y ejecutar (como administrador, eso sí): `usermod -a -G dialout miusuario` (donde

EL MUNDO GENUINO-ARDUINO

"miusuario" lo deberemos cambiar por el nombre de nuestro usuario del sistema, donde el parámetro "-G dialout" especifica el grupo al que lo queremos añadir y donde el parámetro -a indica que tan solo queremos añadir un nuevo grupo al usuario sin eliminarlo de los otros posibles grupos a los que pudiera pertenecer anteriormente). Una vez ejecutado el comando anterior, deberemos reiniciar la sesión para que los cambios tengan efecto.

Breve nota sobre el reconocimiento y uso de dispositivos USB-ACM en Linux

Cada vez que se conecta una placa Arduino UNO a un computador ejecutando un sistema Linux, ocurren dos cosas:

1. La placa (más concretamente el chip ATmega16U2) es reconocida como un dispositivo de tipo "USB ACM" (esto es fácil comprobarlo ejecutando el comando `dmesg` y observando su salida). Los dispositivos USB de tipo ACM ("Abstract Control Modem") tienen la característica de poder establecer con el computador una comunicación serie sencilla y directa a través de USB. En realidad son un subconjunto de un conjunto mayor de dispositivos, los pertenecientes a la clase CDC ("Communications device class"), los cuales a su vez son un pequeño conjunto de todos los posibles chips que cumplen el estándar USB.
2. Se genera automáticamente el fichero `/dev/ttyACM#`, donde la "#" será un número diferente dependiendo de la cantidad de dispositivos del mismo tipo que haya conectados en ese momento (supondremos a partir de ahora que la placa se reconoce como `/dev/ttyACM0`). Igualmente, cuando la placa se desconecta del computador, el fichero `/dev/ttyACM0` desaparece automáticamente. Es fácil comprobar la existencia de este fichero observando la salida de `dmesg` o también ejecutando el comando `ls -l /dev/ttyACM0`.

Si en una distribución Ubuntu o Fedora observáramos la salida del comando "ls" anterior, podríamos comprobar que el propietario de este archivo es el usuario "root" y el grupo propietario de este archivo es el grupo "dialout", los cuales tienen permisos de lectura y escritura sobre el archivo. Esta es precisamente la razón por la que necesitamos incluir nuestro usuario del sistema en el grupo "dialout": para poder tener permisos de lectura y escritura en ese fichero y así poder recibir y enviar datos a través de su dispositivo asociado.

En realidad, el reconocimiento de los periféricos de tipo "USB ACM" por parte de Linux (y la generación del fichero de dispositivo `/dev/ttyACM0` pertinente) es un proceso gestionado por un software llamado "cdc_acm", el cual ya viene integrado por defecto en las distribuciones más extendidas. Así pues, es gracias a la presencia de este software (técticamente es un "módulo de kernel") dentro del sistema Linux que podemos establecer una comunicación serie entre nuestro computador y el dispositivo representado por `/dev/ttyACM0`.

Cualquier sistema Linux (a partir del código fuente)

El código fuente del entorno de programación Arduino se encuentra disponible para descargar en <http://github.com/arduino/Arduino>. No obstante, esta opción solamente se recomienda para usuarios avanzados; para instalar el IDE en un sistema Linux de forma rápida y cómoda se aconseja seguir los pasos descritos en el apartado anterior. Si, aun así, se prefiere compilar el código fuente del IDE para adaptarlo mejor a las características de nuestro computador (o incluso para poder modificar personalmente alguna funcionalidad interna), en el siguiente enlace se pueden encontrar las instrucciones paso a paso para realizar con éxito el proceso: <https://github.com/arduino/Arduino/wiki/Building-Arduino>.

Por otro lado, si se desea participar en el desarrollo del IDE, proponiendo mejoras o informando de errores, en <https://github.com/arduino/Arduino/issues> podemos ver la lista de problemas en el código del IDE pendientes de resolver y su gestión; y en [https://groups.google.com/a/arduino.cc/forum/#!forum/developers](https://groups.google.com/a/arduino.cc/forum/#forum/developers) podemos acceder a la lista de correo de los desarrolladores, donde se realizan consultas sobre el funcionamiento del código y se sugieren ideas para optimizarlo y enriquecerlo, además de discutir sobre el diseño de las diferentes placas y hardware oficial de Arduino.

Windows

Para instalar el IDE de Arduino en Windows, deberemos ir a su página web oficial de descargas: <http://arduino.cc/en/Main/Software>. Allí encontraremos dos enlaces diferentes apuntando a sendos archivos, cada uno de los cuales nos permitirá instalar el IDE de una manera diferente (así que debemos elegir el que más nos convenga). El archivo ejecutable (con extensión .exe) representa un instalador típico de Windows: descargándolo y haciendo clic sobre él (eso sí, siempre que hayamos iniciado sesión con un usuario con privilegios de administrador) pondremos en marcha un asistente que efectuará la instalación del IDE de una forma guiada y muy sencilla. En cambio, descargando –y descomprimiendo– el paquete con extensión .zip, obtendremos una carpeta dentro de la cual no encontraremos ningún instalador sino (entre una estructura de archivos y subcarpetas que no deberemos modificar para nada) un archivo ejecutable llamado "arduino.exe"; haciendo clic dos veces sobre este archivo pondremos el IDE directamente en marcha sin más pasos.

Evidentemente, si optamos por descargar el paquete .zip, es posible mover y guardar la carpeta resultado de la descompresión en el lugar que deseemos de nuestro disco duro (siempre y cuando no alteremos nada de su interior) y, adicionalmente, también podríamos crear –usando la opción adecuada del menú contextual– un acceso directo al ejecutable "arduino.exe" allí donde lo consideremos oportuno (por ejemplo, en el escritorio).

EL MUNDO GENUINO-ARDUINO

Al igual que ocurría con la instalación del IDE en Linux, en Windows tampoco es necesario instalar previamente el "Java Runtime Environment" porque la versión del IDE de Arduino para Windows también lo incorpora dentro de sí mismo. Esto hace que la dependencia de Java no sea ningún problema.

Possible problema: instalación del "driver"

El inconveniente que teníamos en Linux de tener que asignar a nuestro usuario los permisos adecuados para que al ejecutar el IDE este pudiera comunicarse correctamente con la placa Arduino a través del cable USB es solventado automáticamente si optamos por utilizar el asistente de instalación del IDE, ya que durante dicho proceso de instalación es incorporado al sistema el "Arduino USB Driver", componente software responsable precisamente de que dicha comunicación se produzca tal como debe. No obstante, si optamos por descargar el paquete .zip, entonces será necesario realizar la instalación del "Arduino USB Driver" manualmente; para ello, deberemos dirigirnos a la carpeta llamada "drivers" (ubicada dentro de la carpeta resultado de la descompresión) y allí hacer doble clic sobre el ejecutable llamado "dpinst-x86" (si nuestro sistema Windows es de 32 bits) o "dpinst-amd64" (si nuestro sistema Windows es de 64 bits). Para saber de manera rápida si nuestro sistema Windows es de 32 bits o de 64 bits podemos abrir un terminal y escribir el siguiente comando: `wmic os get osarchitecture`, obteniendo directamente la respuesta deseada.

Breve nota sobre el reconocimiento y uso de dispositivos COM en Windows

Cada vez que se conecta una placa Arduino UNO a un computador ejecutando un sistema Windows, esta placa es reconocida como un dispositivo de tipo "COM", y como tal aparece con el nombre de "Arduino UNO (COM#)" dentro del apartado "Puertos (COM y LPT)" del "Administrador de dispositivos" (donde "#" es un número –normalmente 3 o superior, ya que los puertos COM1 y COM2 se reservan para puertos serie hardware reales, no simulados a través de USB–). Este número será importante recordarlo posteriormente para usar nuestra placa correctamente. Para abrir el "Administrador de dispositivos" de Windows, basta con escribir `devmgmt.msc` en su buscador de aplicaciones.

OS X

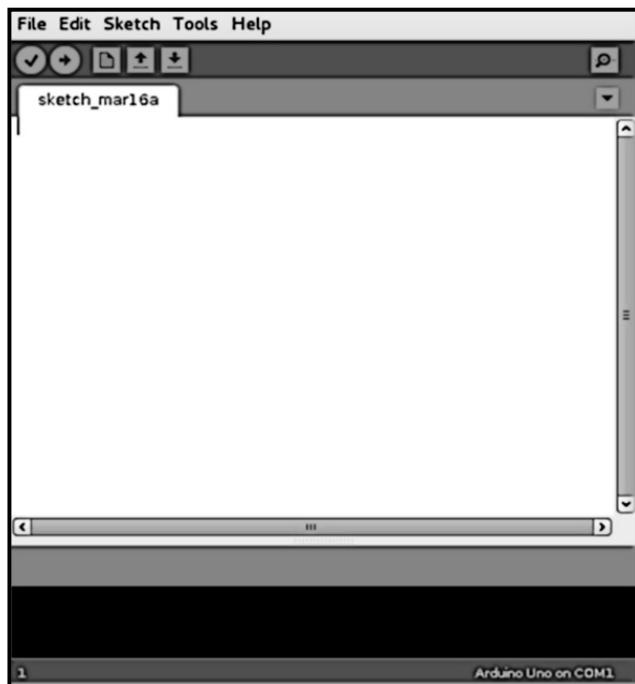
Para instalar el IDE de Arduino en OS X deberemos ir a su página web oficial de descargas: <http://arduino.cc/en/Main/Software>. Allí encontraremos un enlace para descargar la versión del IDE para OS X, que no es más que un archivo comprimido en formato zip. Por lo tanto, lo primero que tendremos que hacer una

vez descargado es descomprimirlo, obteniendo entonces la aplicación "Arduino.app", la cual podremos arrastrar y soltar allí donde nos convenga (en el escritorio, en el lanzador de aplicaciones, etc.). Una vez hecho esto, simplemente haciendo doble clic sobre esta aplicación (y pulsando en el botón "Cancelar" del cuadro emergente, el cual solo aparecerá esta primera vez) ya podremos ver el IDE ante nosotros.

Al igual que ocurría con la instalación del IDE en otros sistemas, en OS X tampoco es necesario instalar previamente el "Java Runtime Environment" porque la versión del IDE de Arduino para OS X también lo incorpora dentro de sí mismo. Esto hace que la dependencia de Java no sea ningún problema.

PRIMER CONTACTO CON EL IDE

Una vez instalado el IDE Arduino (en el sistema que sea), nada más arrancarlo veremos una ventana similar a la siguiente:



Podemos ver que la ventana del IDE se divide en cinco grandes áreas. De arriba abajo son: la barra de menús, la barra de botones, el editor de código propiamente dicho, la barra y la consola de mensajes, y la barra de estado.

EL MUNDO GENUINO-ARDUINO

La zona del IDE donde trabajaremos más tiempo será el editor de código, ya que es allí donde escribiremos nuestros sketches. Otra zona que utilizaremos a menudo será la barra de botones, compuesta por los siguientes elementos:

 **Verify:** este botón realiza dos cosas: comprueba que no haya ningún error en el código de nuestro sketch, y si el código es correcto, entonces lo compila. Este es el primer botón que deberemos pulsar cada vez que deseemos probar cualquier modificación que hagamos en nuestro sketch. (Si no sabes lo que significa "compilar", no te preocupes: por ahora tan solo hay que saber que es un paso imprescindible).

 **Upload:** este botón realiza la misma función que el botón "Verify" pero además, si la compilación es existosa, ejecuta un segundo paso extra: cargar en la memoria del microcontrolador de la placa Arduino el sketch recientemente verificado y compilado. Esto es posible porque al hacer clic en este botón se activa automáticamente el bootloader del microcontrolador (al ejecutarse el "auto-reset" de la placa), así que raras veces será necesario utilizar el método "tradicional" de activar el bootloader mediante la pulsación física del botón de reinicio que viene en la placa.

NOTA: El lector puede estar preguntándose el porqué de la existencia del botón "Verify" si el botón "Upload" ya incluye su funcionalidad. La razón es que muchas veces utilizaremos el botón "Verify" (tal como su nombre indica) para comprobar errores de código, y, solo tras haber verificado que nuestro sketch está correctamente escrito será cuando utilizaremos el botón "Upload" para cargar efectivamente el sketch en la placa.

 **New:** crea un nuevo sketch vacío.

 **Open:** presenta un menú con todos los sketches disponibles para abrir. Podremos abrir tanto nuestros propios sketches como gran cantidad de sketches de ejemplo listos para probar, clasificados por categorías dentro del menú. Estos sketches son muy útiles para aprender; de hecho, en este texto haremos uso de bastantes de ellos, ya que son de dominio público.

 **Save:** guarda el código de nuestro sketch en un fichero, el cual tendrá la extensión ".ino". Podemos guardar estos ficheros donde queramos, pero el IDE Arduino nos ofrece una carpeta específica para ello, la carpeta llamada "Arduino", ubicada dentro de la carpeta personal de nuestro usuario del sistema (en Linux) o dentro de la subcarpeta "Documentos" bajo dicha carpeta personal (en Windows y OS X). Esta carpeta es generada

automáticamente la primera vez que se ejecuta el IDE. Asimismo, dentro de esta carpeta "Arduino" se creará automáticamente una subcarpeta diferente para cada proyecto que desarrollemos, de manera que los sketches de cada proyecto se guarden en subcarpetas separadas y no se mezclen entre sí.



Serial Monitor: abre el "monitor serie". De él hablaremos enseguida.

Podemos ver también que justo debajo del botón de "Serial Monitor" tenemos un botón desplegable desde el cual podemos abrir nuevas pestañas. Tener varias pestañas abiertas a la vez nos puede ser útil cuando tenemos un código tan largo que necesitamos dividirlo en partes para trabajar más cómodamente. Esto es así porque todas las nuevas pestañas abiertas forman parte del mismo proyecto que la primera pestaña original (y por lo tanto, el código escrito en todas ellas es, en global, solo uno) pero el contenido particular de cada una de las pestañas se guarda físicamente en un fichero diferente, permitiendo así una manipulación más sencilla. En estos casos, cuando un proyecto consta de varios ficheros de código fuente (ubicados todos dentro de la misma carpeta), al abrir uno de ellos con el entorno de programación, este detectará la existencia de los demás ficheros y los mostrará automáticamente en sus pestañas correspondientes. Lo más habitual es utilizar pestañas separadas para la definición de funciones, constantes o variables globales (conceptos que estudiaremos en el siguiente capítulo).

Otras acciones triviales que podemos realizar con el botón desplegable son: renombrar la pestaña actual (lo que resulta en el renombramiento del fichero correspondiente), borrar la pestaña actual (lo que resulta en el borrado del fichero correspondiente, ojo), moverse a la pestaña siguiente o anterior, o moverse a una pestaña concreta.

La barra de menú nos ofrece cinco entradas principales: "File", "Edit", "Sketch", "Tools" y "Help". Aclaremos que, posiblemente, estos nombres –y de hecho, todos los elementos del IDE– los verás traducidos ya de entrada al idioma configurado por defecto en tu sistema operativo, pero, para evitar posibles incoherencias, en este libro los seguiremos nombrando mediante su denominación anglosajona, que es la original. Las entradas de la barra de menú muestran acciones adicionales que complementan a las que podemos realizar con la barra de botones. Es interesante notar que los menús son sensibles al contexto, es decir, que solamente los elementos relevantes en un momento determinado (según lo que estemos haciendo) estarán disponibles, y los que no, aparecerán deshabilitados. Algunas de las acciones que podemos realizar mediante los menús son:

EL MUNDO GENUINO-ARDUINO

Menú "File": además de ofrecer acciones estándar como crear un nuevo sketch, abrir uno existente (ya sea seleccionándolo de nuestro disco duro o eligiéndolo directamente de la lista de "más recientes" que muestra el propio menú), guardarlo, cerrarlo, cerrar el IDE en sí, etc., podemos ver también otras acciones interesantes. Por ejemplo, gracias a la entrada "Examples" podemos acceder a los sketches de ejemplo que vienen de serie con el IDE y gracias a la entrada "Sketchbook" podemos acceder a nuestros propios sketches guardados en las diferentes subcarpetas que hay dentro de la carpeta "Arduino". Por otro lado, gracias a la entrada "Preferences", podemos abrir un cuadro emergente que nos ofrece la posibilidad de establecer algunas preferencias del IDE, como por ejemplo la de cambiar la ubicación de la carpeta "Arduino", el idioma del IDE, el tamaño de la fuente de letra o el nivel de detalle (es decir, la "verbosidad") en los mensajes mostrados durante el proceso de compilación y/o carga de los sketches , así como también la opción de activar (o no) las actualizaciones automáticas del IDE, de grabar automáticamente (o no) el sketch al compilarlo o cargarlo, de mostrar en el lateral del código (o no) el número correspondiente a la posición de cada línea, de expandir o contraer (o no) el código interno de las funciones definidas en el sketch (esta opción se llama "code folding" y es muy cómoda cuando queremos "despejar" zonas de nuestros códigos para poder trabajar más cómodo), etc.

Existen muchas preferencias más que no aparecen en el cuadro emergente. Si queremos modificarlas, tendremos que hacerlo "a mano" editando el valor adecuado dentro del fichero de configuración "preferences.txt", que no es más que un fichero de texto que contiene una lista de parejas dato<->valor bastante autoexplicativa. Dependiendo de nuestro sistema operativo, el fichero "preferences.txt" estará ubicado en una carpeta diferente (particular a cada usuario del sistema), carpeta que ya se nos indica en el propio cuadro emergente. Sea como sea, este fichero se ha de modificar cuando el IDE no esté ejecutándose, porque si no, los cambios realizados serán sobrescritos por el propio entorno cuando se cierre.

Menú "Edit": además de ofrecer acciones estándar como deshacer y rehacer, cortar, copiar y pegar texto, seleccionar todo el texto o buscar y reemplazar texto, podemos ver otras acciones interesantes. Por ejemplo, gracias a la entrada "Copy for forum" podemos copiar el código de nuestro sketch al portapapeles de nuestro sistema en una forma que es especialmente adecuada para pegarlo acto seguido directamente en el foro oficial de Arduino (y así poder recibir ayuda de la comunidad). Gracias a la entrada "Copy as HTML" podemos copiar el código de nuestro sketch al portapapeles

de nuestro sistema en una forma que es especialmente adecuada para pegarlo en páginas web genéricas. Otras acciones a tener en cuenta son, por ejemplo, la de comentar/descomentar la porción de texto que tengamos seleccionada, o bien la de aplicarle o quitarle sangría (hablaremos de la importancia de estas dos posibilidades en el siguiente capítulo).

Menú "Sketch": en este menú se ofrece la acción de verificar/compilar nuestro sketch (equivalente al botón "Verify" visto anteriormente), la de cargar el sketch en memoria del microcontrolador (equivalente al botón "Upload"), la de cargar igualmente el sketch pero utilizando en cambio un programador ISP externo (seleccionado previamente de entre una lista de programadores ISP compatibles disponible en el menú "Tools"->"Programmer"), la de compilar el sketch actual y guardar el fichero ".hex" resultante dentro de la carpeta del proyecto, la de abrir la carpeta donde está guardado el fichero ".ino" editado en este momento, la de añadir en una nueva pestaña un nuevo fichero de código a nuestro proyecto y la de importar librerías (de esto último hablaremos en el próximo apartado).

Menú "Tools": en este menú se ofrecen diferentes herramientas variadas, como la posibilidad de autoformatear el código para hacerlo más legible (por ejemplo, sangrando las líneas de texto contenidas dentro de las llaves { y }), la posibilidad de guardar una copia de todos los sketches del proyecto actual en formato zip, la posibilidad de abrir el monitor serie (equivalente al botón "Serial monitor") y/o de abrir el "Serial Plotter" (una "versión gráfica" del "Serial monitor" de cuyas posibilidades hablaremos más adelante), etc. Otras herramientas mucho más avanzadas son la entrada "Programmer" ya comentada (en la cual aparece indicado el programador seleccionado en este momento de entre todos los posibles) o la entrada "Burn bootloader", útil cuando queramos grabar un nuevo bootloader en el microcontrolador de la placa. Mención aparte requieren las entradas "Board" y "Port", que comentaremos más extensamente en los párrafos siguientes.

Menú "Help": desde este menú podemos acceder a varias secciones de la página web oficial de Arduino que contienen diferentes artículos, tutoriales y ejemplos de ayuda. No se necesita Internet para consultar dichas secciones ya que esta documentación se descarga junto con el propio IDE, por lo que su acceso se realiza en local (es decir, "offline"). Se recomienda fervientemente su consulta.

NOTA: Si creáramos dentro de la carpeta de nuestro proyecto (ubicada dentro de la carpeta "Arduino", recordemos) una subcarpeta llamada "docs" y allí incluyéramos la documentación de dicho proyecto en formato de página web, al clicar con el ratón

EL MUNDO GENUINO-ARDUINO

sobre una ruta tal como "`file:///docs/nombrePaginaWeb.html`" escrita dentro de cualquier comentario de cualquier sketch de ese proyecto se nos abriría la página web correspondiente.

La barra y la consola de mensajes informan en el momento de la compilación de los posibles errores cometidos en la escritura de nuestro sketch, además de indicar el estado en tiempo real de diferentes procesos, como por ejemplo la grabación de ficheros "ino" al disco duro, la compilación del sketch, la carga al microcontrolador, etc. Es interesante observar también que cada vez que se realice una compilación exitosa, aparecerá en la consola de mensajes el tamaño que ocuparía el sketch dentro de la memoria Flash del microcontrolador y la cantidad de memoria SRAM que se necesitaría. Esta información es muy valiosa, ya que si, por ejemplo, el tamaño de nuestro sketch estuviera próximo al tamaño máximo permitido (en el caso de la placa Arduino UNO recordemos que es de 32KB), podríamos saberlo y entonces modificar convenientemente el código de nuestro sketch para reducirlo de tamaño y no excedernos del límite.

NOTA: La cantidad de SRAM mostrada en la consola de mensajes tan solo refleja el uso que hacen de ella unos determinados elementos de nuestro sketch (las variables globales), pero no se tienen en cuenta otros (como las variables locales o el posible manejo dinámico de memoria que pudiera existir dentro del código), por lo que esa cantidad indicada solo es una estimación a la baja.

La barra de estado simplemente muestra a su izquierda el número de línea del sketch actual donde en estos momentos está situado el cursor, y a su derecha el tipo de placa Arduino y el puerto serie del computador actualmente utilizado.

Si fuera necesario, podríamos cambiar los colores de los diferentes elementos del IDE (el fondo de las distintas barras, los textos de las diferentes secciones de código, los mensajes de la consola, etc., etc.) modificando convenientemente un fichero de texto llamado "theme.txt" ubicado dentro de la carpeta "lib/theme" (bajo la carpeta principal de Arduino). En ese fichero aparece una lista de los diferentes elementos del IDE, cada uno asociado a un determinado color, el cual ha de especificarse mediante el formato "RGB hexadecimal" (<http://www.javascripter.net/faq/rgbtohex.htm>). Por otro lado, en la misma carpeta "lib/theme" podemos observar la presencia de diferentes imágenes que representan los diferentes botones del IDE (estas imágenes podrían ser reemplazadas por otras –que tuvieran el mismo tamaño y formato– para conseguir así un cambio estético todavía más radical) y otros ficheros, como "formatter.conf" (responsable de establecer las características por defecto de la opción "Tools->Auto Format" –por ejemplo, allí se define el número de espacios usados en una sangría de texto–) o "keywords.txt" (responsable del coloreado especial de las palabras propias del lenguaje Arduino).

HERRAMIENTAS EXTRA INTEGRADAS EN EL IDE

Las librerías y el "Library Manager"

Concepto de librería

Una librería (mala traducción del inglés "library", que significa "biblioteca") es un conjunto de instrucciones de un lenguaje de programación agrupadas de una forma coherente. Podemos entender una librería como un "cajón" etiquetado que guarda unas determinadas instrucciones relacionadas entre sí. Así, tenemos el "cajón" de las instrucciones para controlar motores, el "cajón" de las instrucciones para almacenar datos en una tarjeta de memoria, etc.

Las librerías sirven para proveer funcionalidad extra (manipulando datos, interactuando con hardware, etc.) pero también para facilitar el desarrollo de nuestros proyectos, porque están diseñadas para que a la hora de escribir nuestro programa no tengamos que hacer "la faena sucia" de conocer todos los detalles técnicos sobre el manejo de un determinado hardware, o ser un experto en la configuración de determinado componente, ya que las librerías ocultan esa complejidad.

Además, la organización del lenguaje en librerías permite que los programas resulten más pequeños y sean escritos de una forma más flexible y modular, ya que solo están disponibles en cada momento las instrucciones ofrecidas por las librerías utilizadas en ese instante.

Físicamente una librería no es más que una carpeta (generalmente comprimida en forma de paquete .zip) en cuyo interior se hallan unos determinados ficheros. Concretamente, los ficheros que forman una librería típica son: los que contienen el propio código fuente de esa librería (los cuales generalmente tendrán la extensión ".cpp" –por ser código fuente de tipo C/C++– y/o ".h" –por ser cabeceras también C/C++– y deberán estar ubicados en una subcarpeta llamada "src" dentro de la librería); los ficheros que contienen códigos Arduino de ejemplo, pensados para ser mostrados bajo el menú "File->Examples" del IDE (estos ficheros generalmente tendrán la extensión ".ino" y deberán estar ubicados en una subcarpeta llamada "examples" dentro de la librería); un fichero de texto llamado "keywords.txt" que indica al IDE Arduino qué nuevas palabras especiales ha de colorear cuando se escriban en nuestro sketch; un fichero de texto llamado "README.adoc" con la información que el desarrollador considere relevante (normalmente aparece una descripción más o menos extensa de la librería, diversos enlaces de interés, el texto

EL MUNDO GENUINO-ARDUINO

de la licencia...) y, finalmente, otro fichero de texto llamado "library.properties" que contiene información sobre la propia librería en sí (como su nombre, versión, autor, descripción, sitio web, arquitectura compatible... para saber más sobre este fichero y su formato, recomiendo leer <https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification>). Opcionalmente también puede aparecer dentro de la librería una subcarpeta llamada "extras", pensada para que el desarrollador pueda incluir allí lo que estime oportuno (generalmente, documentación extra).

El lenguaje Arduino incorpora por defecto una serie de librerías oficiales que iremos estudiando a lo largo de este libro (son las que aparecen listadas aquí: <http://arduino.cc/en/Reference/Libraries>), pero también ofrece la posibilidad de descargar y utilizar librerías creadas por terceros (hay literalmente decenas) que amplían la funcionalidad del propio lenguaje y permiten que la placa Arduino se adapte a multitud de escenarios diferentes (en este libro, de hecho, conoceremos una gran variedad de ellas). Lo que no estudiaremos, sin embargo, es cómo programar nosotros una librería propia, ya que para ello es necesario tener conocimientos relativamente avanzados del lenguaje de programación C y C++.

Para poder utilizar en nuestro código instrucciones pertenecientes a una librería de terceros, deberemos realizar dos pasos:

- I. Instalar en nuestro computador la librería en cuestión para que pueda ser reconocida por el IDE Arduino. Este paso (que solo se necesita hacer una vez por cada librería) puede realizarse de dos maneras diferentes: o bien "manualmente", o bien a través de una herramienta llamada "Library Manager". Consultar los siguientes apartados para más detalles.
- II. Importar esa librería dentro de cada sketch que queramos que haga uso de las instrucciones contenidas en ella. Consultar el apartado "Cómo importar librerías" de más adelante para más detalles.

Para poder utilizar en nuestro código instrucciones pertenecientes a una librería oficial de Arduino, tan solo será necesario realizar el segundo de los pasos anteriores (la "importación") porque, tal como ya hemos dicho, las librerías oficiales ya se instalan por defecto junto al IDE Arduino. En este caso, el lector puede dirigirse directamente al apartado "Cómo importar librerías".

Cómo instalar librerías (de terceros) manualmente

Los pasos a seguir son estos:

1. Descargar la librería deseada (que deberá estar disponible en forma de paquete comprimido .zip) desde su propio sitio web.

NOTA: Actualmente es muy habitual que el sitio web de muchas librerías Arduino esté bajo el amparo del servicio GitHub (<https://github.com>): en ese caso, el paquete .zip deseado se puede descargar pulsando simplemente sobre el botón "Download ZIP" que aparece bajo la columna derecha de la página principal de la librería en cuestión.

2. Sin descomprimir el paquete recién descargado, abrir el IDE y acceder a su opción del menú "Sketch->Include library->Add library" para seleccionar dicho paquete .zip de nuestro disco duro. Y ya está. Podremos comprobar que la nueva librería se haya instalado correctamente si vemos que aparece dentro de la lista de librerías disponibles (esta lista se muestra bajo el menú "Sketch->Include library").

NOTA: La opción "Add library" también es capaz de importar una librería si seleccionamos, en vez del paquete zip, la carpeta resultante de haber descomprimido previamente dicho paquete. Así pues, podemos optar por descomprimir o no el paquete .zip: a efectos prácticos de importación es irrelevante.

En realidad, lo único que hace la opción "Add library" del IDE es descomprimir el paquete .zip y copiar la carpeta resultante –la cual, por cierto, da nombre a la librería en cuestión, nombre que no puede contener ni el signo "-" ni el signo "_"– dentro de una subcarpeta llamada "libraries" ubicada (si no existe, la crea) dentro de la misma carpeta donde guardamos todos nuestros sketches (es decir, por defecto dentro de la carpeta "Arduino"). Por tanto, este procedimiento podría ser realizado manualmente, pero no hay duda que la opción "Add library" facilita mucho las cosas.

NOTA: La razón de no copiar las librerías de terceros en la misma carpeta donde están situadas las librerías oficiales (que es una carpeta también llamada "libraries" pero ubicada dentro del directorio de instalación del IDE) es para evitar que en una posible actualización del IDE, estas librerías extra sean eliminadas.

Cómo instalar librerías (de terceros) usando el "Library Manager"

Los pasos a seguir son estos:

1. Abrir el IDE y acceder a su opción del menú "Sketch->Include library->Manage libraries...". Al hacer esto se mostrará un cuadro emergente, el "Library Manager".
2. Dentro del "Library Manager" podemos buscar la librería deseada ya sea escribiendo su nombre (si lo conocemos) en el cuadro de texto que hay disponible para ello o bien encontrándola en la lista de resultados tras

EL MUNDO GENUINO-ARDUINO

filtrar por categorías ("Audio", "Sensors", "Communications", etc.) y/o por tipos ("Updatable" –actualizables, si ya fueron instaladas previamente–, "Arduino" –oficiales–, etc.).

3. Una vez seleccionada la librería a instalar, aparecerá el botón "Install" para proceder a su descarga de Internet y su posterior instalación en nuestro computador (algunas veces también podremos elegir la versión de la librería entre varias posibles; por defecto el "Library Manager" siempre opta por la más moderna).

NOTA: En el caso de que para acceder a Internet estemos obligados a utilizar un servidor proxy corporativo (en entornos domésticos no deberemos preocuparnos de esto), el IDE Arduino ofrece la posibilidad de especificarlo mediante la pestaña "Network" del cuadro "Preferences".

4. Tras finalizar el paso anterior, en la lista de resultados del "Library Manager" deberá aparecer la etiqueta "INSTALLED" junto al nombre de la librería recién instalada. Ya se podrá entonces cerrar el "Library Manager". En cualquier momento podremos comprobar que, efectivamente, la nueva librería se ha instalado de forma correcta si accedemos al menú "Sketch->Include library" y vemos que aparece dentro de la lista de librerías disponibles.

Para que una librería aparezca listada en el "Library Manager", el desarrollador de esta ha de haber seguido una serie de pasos (descritos completamente en el documento <https://github.com/arduino/Arduino/wiki/Library-Manager-FAQ>) entre los que se incluyen alojar la librería en un sitio web como GitHub o similar (siguiendo unas determinadas pautas) y solicitar formalmente al Arduino Team la inclusión de la librería en el "Library Manager". Esto significa que no todas las librerías que podamos encontrar en el vasto ecosistema Arduino están disponibles en el "Library Manager": solo aquellas cuyos desarrolladores han seguido los pasos mencionados son las que allí aparecen. Por tanto, si una librería concreta no se muestra en el "Library Manager" no pasa nada: simplemente tendremos que instalarla de forma "manual".

Las librerías instaladas mediante el "Library Manager" se guardan en una subcarpeta dentro de la carpeta donde está ubicado el archivo de preferencias del IDE ("preferences.txt"), carpeta que es particular a cada usuario y diferente en cada sistema operativo.

Cómo importar librerías

Ya se trate de una librería oficial como de una librería de terceros –esto es indiferente–, para importarla en nuestro sketch tan solo debemos hacer un paso: ir al

menú "Sketch->Include library" y seleccionar la librería deseada de la lista de librerías disponibles que allí aparece.

Se puede observar que al hacer este paso, al inicio del código de nuestro sketch se añadirá automáticamente una o más líneas de apariencia similar a la siguiente: `#include <nombreLibreria.h>`. Esta/s línea/s (que también podríamos haber escrito a mano perfectamente sin utilizar la opción del menú del IDE) son las que realmente permite/n utilizar en nuestro sketch las instrucciones pertenecientes a la librería indicada. Esto significa que si esta/s línea/s no apareciera/n al comienzo del código, la librería correspondiente no estaría siendo importada y, por tanto, las instrucciones pertenecientes a ella que pudieran aparecer en nuestro sketch no serían reconocidas, provocando un error de compilación.

Fijarse que (a diferencia del resto de instrucciones del lenguaje Arduino, tal como veremos en el próximo capítulo) no se escribe ningún punto y coma al final de la línea `#include`. Esto es debido a que la palabra especial `#include` en realidad no es una instrucción sino que es lo que se llama "una directiva del preprocesador". Afortunadamente, no nos es necesario conocer este extremo: con recordar que al final de la línea `#include` no se ha de escribir punto y coma ya será suficiente.

Hay que tener en cuenta que las librerías que utilicemos también se cargan en la memoria Flash del microcontrolador, por lo que aumentan el tamaño de nuestro sketch. Así que es buena idea eliminar las líneas `#include` correspondientes a librerías no utilizadas, si hubiera.

El "Boards Manager"

Con el IDE Arduino recién instalado tan solo podremos utilizar placas Arduino basadas en arquitectura AVR (es decir, todas menos la Due y la Zero). Esto es así porque, para conseguir que la descarga del IDE sea más ligera y rápida, no se han incluido en él los ficheros necesarios para trabajar con la arquitectura ARM (arquitectura que en el mundo Arduino actualmente es bastante menos habitual que la AVR). Por tanto, en el caso de que queramos compilar y cargar sketches para placas Due y/o Zero, será necesario, tras instalar el IDE de forma estándar, descargar esos ficheros adicionales (o dicho más técnicamente, descargar el "core SAM3X" y/o el "core SAMD", respectivamente) mediante una herramienta llamada "Boards Manager". Concretamente, para instalar un *core* a través del "Boards Manager", los pasos a seguir son estos:

1. Abrir el IDE y acceder a su opción del menú "Tools->Board->Boards Manager". Al hacer esto se mostrará un cuadro emergente, el "Boards Manager", precisamente.

EL MUNDO GENUINO-ARDUINO

2. Dentro del "Boards Manager" veremos una lista de los *cores* ya instalados (señalados mediante la palabra "INSTALLED"; por defecto este solo será el caso del *core* AVR) y de los disponibles para instalar (el *core* "SAM3X", el *core* "SAMD" y algunos más de terceras empresas cuyas placas han sido certificadas por Arduino, como la Intel Galileo o la Intel Edison, entre otras). Incluso podemos buscar el *core* deseado ya sea escribiendo su nombre (si lo conocemos) en el cuadro de texto que hay disponible para ello o bien encontrándolo en la lista de resultados tras filtrar por tipos ("Arduino" –los *cores* oficiales–, "Arduino Certified" –los de terceros pero certificados–, etc.).
3. Una vez seleccionado el *core* a instalar, aparecerá el botón "Install" para proceder a su descarga de Internet y su posterior instalación en nuestro computador. También podremos elegir la versión del *core* de entre las varias posibles que aparecen listadas en un desplegable; por defecto el "Boards Manager" siempre opta por la versión más moderna.

NOTA: En el caso de que para acceder a Internet estemos obligados a utilizar un servidor proxy corporativo (en entornos domésticos no deberemos preocuparnos de esto), el IDE Arduino ofrece la posibilidad de especificarlo mediante la pestaña "Network" del cuadro "Preferences".

4. Tras finalizar el paso anterior, en la lista de resultados del "Boards Manager" deberá aparecer la etiqueta "INSTALLED" junto al nombre del *core* recién instalado. Ya se podrá entonces cerrar el "Boards Manager". En cualquier momento podremos comprobar que, efectivamente, el nuevo *core* se ha instalado de forma correcta (y, por tanto, que ya podremos trabajar en el IDE con la placa o placas asociadas) si accedemos al menú "Tools->Board" y vemos precisamente que la/s placa/s que incorpora/n el *core* recién instalado aparece/n dentro de la lista de placas disponibles.

El "Boards Manager" nos permite incluso actualizar un *core* ya instalado en nuestro computador si detecta que existe una nueva versión disponible para descargar. Para ello tan solo debemos seleccionar "Updatable" del desplegable de tipos para que aparezcan en la lista los *cores* candidatos a ser actualizados y seguidamente elegir el deseado. Afortunadamente, no será necesario tener que acordarnos de entrar en el "Boards Manager" cada vez que queramos ver si hay disponible una actualización: el IDE ya está programado para que nada más ser iniciado compruebe automáticamente si existe la posibilidad de actualizar algún *core*, notificándonoslo (mediante un cuadro emergente) si se diera el caso.

Tal como ya se ha comentado en párrafos anteriores, el "Boards Manager" no solamente muestra los *cores* oficiales de Arduino, sino que también algunos otros *cores* de terceros vinculados a placas que no forman parte del ecosistema Arduino pero que, sin embargo, pueden ser desarrolladas utilizando el mismo IDE. De hecho, existen muchos otros *cores* de terceros que pueden ser añadidos al "Boards Manager" más allá de los que ya vienen "de serie", permitiendo así que el IDE Arduino sea mucho más que una simple aplicación constreñida a las placas Arduino oficiales convirtiéndose en un entorno de desarrollo común, homogéneo y global a muchos otros tipos de placa. La lista oficial de *cores* de terceros soportados por el "Boards Manager" (y, por ende, descargables y utilizables por el IDE Arduino) se puede consultar en <https://github.com/arduino/Arduino/wiki/Unofficial-list-of-3rd-party-boards-support-urls>; en esta lista se encuentran tanto *cores* pertenecientes a placas propias de varios fabricantes conocidos (Adafruit, Sparkfun, Seeedstudio, Akafugu, ESP8266...) como a placas menos populares pero igualmente interesantes. Para que cualquiera de los *cores* anteriores pueda ser reconocido por el "Boards Manager") deberemos escribir dentro del cuadro de texto "Additional Boards Manager URLs" las URLs completas (es decir, las direcciones HTTP/HTTPS enteras) de los archivos *package_XXX_index.json* correspondientes a cada uno de los *cores* deseados (cada URL ha de ir en una línea diferente) y reiniciar el IDE: tras ello ya debería aparecer dentro de la lista de *cores* mostrada por el "Boards Manager" los nuevos *cores* listos para ser descargados e instalados (apareciendo, en el caso de proceder a su instalación, las placas vinculadas dentro del menú "Tools->Boards" listas para ser elegidas).

El fichero *package_XXX_index.json* es un fichero de tipo texto escrito en formato JSON que el desarrollador del *core* proporciona para que el "Boards Manager" tenga la información necesaria para saber localizar y descargar el *core* asociado (que no deja de ser una carpeta comprimida en formato zip). El nombre de este fichero ha de ser siempre del estilo "*package_nombrefabricante_index.json*" y su contenido ha de tener una estructura interna concreta, explicada en https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.6.x---package_index.json-format-specification.

El *core*, por su parte, ha de reunir una serie de características determinadas para que sea válido y pueda ser reconocido por el IDE (tras haber sido descomprimido) sin problemas. Estas características están detalladas en el documento <https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5-3rd-party-Hardware-specification>. Entre ellas podemos destacar la obligatoriedad de contener todos sus ficheros dentro de una subcarpeta para cada arquitectura soportada, encontrándose todas ellas dentro de una carpeta propia del fabricante ubicada a su

EL MUNDO GENUINO-ARDUINO

vez dentro de la carpeta "hardware" existente dentro de la carpeta "Arduino" de nuestro usuario (es decir, si el fabricante se llamara "pepe" y ofreciera una placa AVR y otra SAM, los ficheros de su *core* debería estar contenidos dentro de las carpetas "avr" y "sam", respectivamente, y ambas hallarse dentro de la carpeta "pepe", la cual debería estar ubicada dentro de "Arduino/hardware"); otra característica obligatoria de los *cores* es que deben contener, para cada arquitectura, como mínimo tres ficheros: "platforms.txt" (que contiene definiciones para la arquitectura utilizada: herramientas de compilación, parámetros del proceso de compilación, herramientas usadas para la carga de sketches, etc.); "boards.txt" (que contiene definiciones de las diferentes placas específicas ofrecidas para esa arquitectura: nombre de la placa, parámetros más concretos para compilar y cargar sketches, bootloader utilizado, etc.; cada una de las placas allí definidas aparecerán, una vez instalado el *core*, bajo el menú "Tools->Boards", listas para ser elegidas) y "programmers.txt" (que contiene definiciones para programadores externos, generalmente usados para grabar bootloaders o sketches sin usar estos). Además de estos tres ficheros, cada arquitectura del *core* contendrá otras subcarpetas, como "firmwares", "bootloaders", "libraries", etc.

NOTA: En el archivo "boards.txt" no solamente se encuentran definiciones de placas, sino también de diferentes bootloaders para una misma placa. Esto permite que podamos seleccionar uno u otro en el menú "Tools-Boards" según nos interese.

Las cores instalados mediante el "Boards Manager" se guardan en una subcarpeta dentro de la carpeta donde está ubicado el archivo de preferencias del IDE ("preferences.txt"), carpeta que es particular a cada usuario y diferente en cada sistema operativo.

El "Serial Monitor" y otros terminales serie

El "Serial monitor" es una ventana del IDE que nos permite desde nuestro computador enviar y recibir datos textuales a la placa Arduino usando el cable USB (más exactamente, mediante una conexión serie asíncrona). Para enviar datos, simplemente hay que escribir el texto deseado en la caja de texto que aparece en su parte superior y clicar en el botón "Send" (o pulsar Enter). Aunque, evidentemente, no servirá de nada este envío si la placa no está programada con un sketch que sea capaz de obtener estos datos y procesarlos. Por otro lado, los datos recibidos provenientes de la placa serán mostrados en la sección central del "Serial monitor".

Es importante elegir mediante la caja desplegable de la parte inferior derecha del "Serial monitor" la misma velocidad de transmisión (medida en "baudios") que la que se haya especificado en el sketch ejecutado en la placa, porque si no, los caracteres transferidos no serán reconocidos correctamente y la comunicación no tendrá sentido.

De todas formas, no solo mediante el "Serial monitor" podemos comunicarnos mediante una conexión serie (a través de USB) con la placa. Se puede utilizar cualquier otro programa que permita enviar y recibir datos a través de conexiones de este tipo. Estos programas se suelen denominar "terminales serie". En los repositorios de Ubuntu y Fedora (y de la mayoría de distribuciones Linux) podemos encontrar varios, como **Gtkterm**, **Cutecom**, **Picocom** o **Minicom**, entre otros. En Windows podemos usar otros programas similares, como **TeraTerm** (<http://ttssh2.sourceforge.jp>), **Putty** (<http://www.putty.org>) o **Terminalbpp** (<https://sites.google.com/site/terminalbpp>), entre otros. En OS X una buena aplicación es **CoolTerm** (<http://freeware.the-meiers.org>) o también **ZTerm** (<http://www.dalverson.com/zterm>).

En cualquiera de los programas anteriores no es necesario realizar ninguna configuración especial para que funcionen correctamente con nuestra placa: tan solo se requiere especificar el puerto serie usado para establecer la comunicación (esto es, /dev/ttyACM0 en Linux, COM3 en Windows, etc.) y la velocidad –en baudios– a la que se efectuará la transmisión de datos (dato que dependerá del sketch concreto que en ese momento esté ejecutándose en la placa). Por ejemplo: escribir el comando `picocom /dev/ttyACM0 -b 9600` es todo lo que necesitamos en un sistema Linux para establecer comunicación (a 9600 baudios) con una placa accesible a través del puerto /dev/ttyACM0. Una vez establecida dicha comunicación, ya tendremos un canal serie por donde podremos recibir (o enviar) datos de (o hacia) la placa, siempre y cuando, insistimos, esta haya sido previamente programada para ello. Para cerrar ese canal y salir del programa, en `picocom` es necesario pulsar las combinaciones de teclas CTRL+a y, a continuación, CTRL+x. Otros programas son similares: para un análisis más exhaustivo se puede consultar el artículo <https://learn.sparkfun.com/tutorials/terminal-basics>

NOTA: Ya sabemos que para definir completamente las características de una comunicación serie asíncrona no basta con establecer la velocidad de transmisión, sino que también es necesario que ambos extremos acuerden (para que la información enviada o recibida entre ellos pueda interpretarse con sentido) otros aspectos, como son el número de "bits de datos", el de "de paridad", el "de inicio", etc. La buena noticia es que todas las placas Arduino funcionan con los mismos valores de estas características, los cuales son, precisamente, los valores utilizados por defecto en la gran mayoría de terminales serie si no son indicados explícitamente. En concreto, estos valores son: 8 "bits de datos", ningún "bit de paridad", 1 "bit de parada" y sin control de flujo (o más abreviadamente, 8-N-1-N).

Ejecución del "auto-reset" al abrir el "Serial Monitor"

Al abrir el "Serial monitor" (o cualquier otro programa "terminal serie") es posible que la placa Arduino que tengamos conectada al puerto USB de nuestro ordenador se reinicie automáticamente, volviéndose entonces a ejecutar de nuevo

EL MUNDO GENUINO-ARDUINO

desde el principio el sketch grabado en ella. Esto solo ocurre, no obstante, si nuestra placa es de un determinado modelo: más concretamente si es de cualquiera que no sea ni el modelo Zero, ni el modelo Due –al usar su "Native USB Port"– ni los que incorporan el microcontrolador ATmega32U4 (esto es, los modelos Micro, Yún y LilyPadUSB); dichas excepciones no tienen este comportamiento de "auto-reset" al abrir el "Serial monitor", de manera que en ellas el sketch continúa ejecutándose "como si nada" mientras establecemos la conexión a través del canal serie.

En el caso de utilizar un modelo de placa afectado por este auto-reinicio provocado al empezar desde el exterior una comunicación a través del canal serie (UNO, Pro, Mega...) debemos tener en cuenta que, como en cualquier otro proceso de inicio de una placa, durante aproximadamente medio segundo su bootloader se pondrá en marcha. Por tanto, durante ese medio segundo, será el bootloader quien intercepte los primeros bytes de datos enviados desde el computador a la placa justo tras el establecimiento de la conexión USB. El problema de esto es que, aunque el bootloader no hará nada con estos bytes porque los ignorará, estos ya no llegarán al destino deseado (es decir, al programa grabado en el microcontrolador propiamente dicho), porque este se pondrá en marcha con ese retraso de medio segundo. Por lo tanto, hay que tener muy en cuenta que si nuestro programa grabado en el microcontrolador está pensado para recibir nada más iniciarse datos provenientes de nuestro computador vía USB, nos tendremos que asegurar de que el software encargado de enviarlos se espere un segundo después de abrir la conexión USB para proceder al envío.

El auto-reinicio es un comportamiento que durante el desarrollo de nuestros sketches puede ser bastante conveniente para ir depurando errores pero a la hora de montar un proyecto estable puede no ser tan buena idea: imaginemos, por ejemplo, que tenemos una placa UNO funcionando como servidor web (esto lo estudiaremos en el último capítulo) y en un momento determinado conectamos con ella a través de su canal serie: durante unos cuantos segundos dejará de ofrecer páginas web a los clientes porque estará reiniciándose. Si no queremos que esto ocurra (esto es: si queremos poder conectarnos a su canal serie sin alterar su funcionamiento continuo habitual) debemos deshabilitar el "auto-reset". Para ello, lo más sencillo es colocar una patilla de un condensador pequeño ($10\mu F$ ya va bien) en el pin-hembra "RESET" de nuestra placa y la otra en un pin-hembra "GND": haciendo esto, el condensador "se tragará" el pulso de reinicio.

NOTA: La colocación del condensador solamente se puede hacer una vez que la placa ya está en pleno funcionamiento: si se coloca antes de iniciarla, no podrá arrancar. Por otro lado, cada vez que deseemos volver a cargar un nuevo sketch, deberemos retirar el condensador.

En el caso contrario (es decir, en el caso de trabajar con una placa que no se auto-reinicia al recibir a través de USB la señal de apertura del canal serie –o más técnicamente, de un canal de tipo USB-CDC– y querer que sí se auto-reinicie) la solución pasa por modificar nuestros sketches. En vez de forzar un auto-reinicio al detectar la conexión serie (posibilidad que el lenguaje Arduino no ofrece) lo que podemos hacer (atención, este truco no funciona en placas Arduino Zero) es mantener nuestro sketch a la espera "sin hacer nada" justo tras haberlo arrancado (es decir, bloquearlo al principio de la sección `setup()`) y desbloquearlo solamente en el momento de detectar dicha conexión. Para ello simplemente debemos escribir dentro de la sección `setup()` de código dos líneas: `Serial.begin(9600);` –para abrir el canal serie de la forma habitual– y `while(!Serial){};` –para mantener a la placa en estado latente hasta que se abra la conexión USB-CDC desde el exterior– (en el capítulo siguiente se comprenderá todo esto mucho mejor). Haciendo esto, cuando abramos por primera vez el "Serial Monitor" (o similar), el comportamiento de las placas Yún, Micro, etc., será equivalente al de los modelos con "auto-reinicio"; así pues, entre otros efectos, podremos ver los datos enviados por la placa al computador durante sus primeros segundos de funcionamiento tras el desbloqueo (mediante líneas `Serial.print()` o similares dentro de la propia `setup()`, por ejemplo) sin perderlos, como pasaría tras un reinicio estándar.

COMPROBACIÓN DEL CORRECTO FUNCIONAMIENTO DEL IDE

Una vez hemos conectado mediante el cable USB nuestra placa recién adquirida a nuestro computador, lo primero que deberíamos ver es que el LED etiquetado como "ON" se enciende y se mantiene encendido de manera continua siempre. Tras comprobar esto, ya podemos poner en marcha el IDE de Arduino. Una vez abierto, no obstante, antes de poder empezar a escribir una sola línea debemos asegurarnos de que:

1. El IDE reconozca el tipo de placa Arduino conectada en este momento al computador (UNO, Mega, etc.).
2. El IDE reconozca qué puerto serie de nuestro computador es el usado por este para comunicarse vía USB con la placa.

Estos dos datos son detectados por el IDE automáticamente al ponerse en marcha, por lo que, si todo va bien, nosotros no deberíamos hacer nada. Para comprobar que la detección se haya realizado correctamente, deberemos fijarnos en la información mostrada a la derecha de la barra de estado (es decir, la más inferior) del IDE, la cual representa, precisamente, el modelo de placa detectado y el puerto serie utilizado para comunicarse con ella.

EL MUNDO GENUINO-ARDUINO

Si, por alguna razón, necesitáramos establecer manualmente el modelo de placa con el que trabajar y el puerto serie a emplear, el IDE ofrece la posibilidad de ello. Para lo primero, simplemente debemos ir al menú "Tools"->"Boards" y seleccionar de la lista que aparece el modelo de placa deseado. Fijarse que hay modelos de placa que aparecen varias veces según si pueden venir con diferentes variantes de microcontrolador. Para lo segundo, debemos ir al menú "Tools"->"Port" y elegir el puerto serie adecuado. Dependiendo del sistema operativo utilizado, en ese menú aparecerán opciones diferentes. Las más comunes son:

Sistemas Linux: aparecen los ficheros de dispositivo (generalmente /dev/ttyACM#).

Sistemas Windows: aparece una lista de puertos COM#. Si aparecen varios puertos COM y no se sabe cuál es el correspondiente a la placa Arduino, lo más sencillo es desconectarla y reabrir el menú: la entrada que haya desaparecido será la que buscamos; conectando otra vez la placa podremos seleccionarla sin problemas.

Sistemas OS X: aparecen los ficheros de dispositivo (generalmente /dev/tty.usbmodem#). Es posible que en el momento de conectar una placa UNO o Mega se muestre una ventana emergente informando de que se ha detectado un nuevo dispositivo de red. Si ocurre esto, basta con clicar en "Network preferences..." y cuando aparezca la nueva ventana, simplemente clicar en "Apply". La placa se mostrará como "Not configured", pero funcionará correctamente por lo que ya se puede salir de las preferencias del sistema.

Ahora solo nos falta comprobar que todo funcione correctamente ejecutando algún sketch de prueba. Como todavía no sabemos escribir ninguno por nuestra cuenta, utilizaremos un sketch de los que vienen como ejemplo. Concretamente, el sketch llamado "Blink". Para abrirlo, debemos ir al menú "File"->"Examples"->"01.Basics"->"Blink". Ahora no nos interesa estudiar el código que aparece: simplemente hemos de pulsar en el botón "Verify" y seguidamente en el botón "Upload". Siempre deberemos seguir esta pauta con todos los sketches: una vez queramos probar cómo funcionan, debemos pulsar primero en "Verify" (para verificar que el código no contenga errores) y seguidamente en "Upload" (para cargarlo, ya compilado, en el microcontrolador). Justo después de haber pulsado este último botón, ocurre que:

1. Primero parpadeará muy rápidamente el LED etiquetado como "L", indicando que la placa se ha reseteado y, por lo tanto, que se está ejecutando el bootloader.

2. Seguidamente los LEDs de la placa etiquetados como "RX" y "TX" parpadearán rápidamente varias veces, indicando que el sketch está llegando a la placa y, por tanto, que está siendo recibido por el bootloader y cargado en la memoria Flash del microcontrolador. En el IDE deberemos observar mensajes informando del estado del proceso, y de su finalización exitosa.
3. Finalmente, una vez acabada la carga, lo que debería ocurrir es que el LED etiquetado como "L" empezara a parpadear de forma periódica, ya para siempre. Si ocurre eso, ¡felicitaciones!: nuestra placa funciona perfectamente y nuestro computador la puede programar sin problemas.

USO DEL IDE EN EL INTÉPRETE DE COMANDOS

Es posible ejecutar el IDE desde un intérprete de comandos; para ello, tras abrir nuestro intérprete favorito (en Linux dos muy populares son *gnome-terminal* o *konssole* pero hay muchísimos más, Windows nos ofrece *cmd.exe* o *powershell.exe* y OS X, *Terminal.app*), primero deberemos situarnos mediante el comando `cd` dentro de la carpeta donde se halla el IDE (es decir, la carpeta generada tras la descompresión del paquete descargado de la web de Arduino). Una vez allí, para mostrar el IDE y empezar a trabajar en él, en sistemas Linux deberemos escribir `./arduino` (importante no olvidarse del punto y la barra); en sistemas Windows, `arduino.exe` y en sistemas OS X, `Contents/MacOS/Arduino`.

En vez de poner en marcha el IDE vacío, podríamos abrir un sketch para empezar a trabajar directamente en él si, tras escribir la orden adecuada (`./arduino` en Linux, `arduino.exe` en Windows, etc.), indicamos la ruta del fichero ".ino" donde está guardado el sketch en cuestión. Por ejemplo, en Linux podríamos escribir `./arduino ~/Arduino/misketch.ino` para abrir el sketch "misketch.ino" ubicado en la carpeta "Arduino". Si el proyecto constara de varios sketches separados, solo indicando uno de ellos ya será suficiente para abrir el resto automáticamente.

Otra funcionalidad ofrecida por el IDE de Arduino que está disponible desde un intérprete de comandos es la posibilidad tanto de compilar ("verify") como de cargar en memoria del microcontrolador ("upload") un determinado sketch (el cual, eso sí, deberá haber sido escrito y guardado previamente usando un editor de texto cualquiera). Primero será necesario, no obstante, especificar el modelo de placa a utilizar y el puerto serie donde está conectada. Esto se hace escribiendo tras la orden adecuada (`./arduino` en Linux, `arduino.exe` en Windows, etc.) lo siguiente: `--board fabricante:arq:placa --port puerto`, donde *fabricante* es el identificador del fabricante de la placa (en placas Arduino oficiales, este identificador siempre será

EL MUNDO GENUINO-ARDUINO

"arduino"), *arq* es la arquitectura del microcontrolador (si usamos placas basadas en microcontroladores AVR deberemos indicar el valor "avr"; si usamos la placa Arduino Due deberemos indicar el valor "sam" y si usamos la placa Arduino Zero deberemos indicar el valor "samd"), *placa* es el nombre específico de la placa a utilizar ("uno", "mega", etc.) y *puerto* es /dev/ttyACM# en Linux, COM# en Windows y /dev/tty.usbmodem# en OS X.

NOTA: Para conocer todos los posibles valores de fabricantes, basta con conocer el nombre de las subcarpetas ubicadas dentro de la carpeta "hardware" existente bajo la carpeta donde se halla el IDE; para conocer todos los posibles valores de arquitecturas, basta con conocer el nombre de las subcarpetas ubicadas dentro de las subcarpetas de cada fabricante; para conocer todos los posibles valores de placas, basta con leer el archivo "boards.txt" ubicado dentro de las subcarpetas de cada arquitectura.

Una vez especificado el modelo de placa y el puerto serie a utilizar, ahora sí: cada vez que queramos compilar un sketch (y, por tanto, detectar si hay algún fallo en nuestro código) bastará con escribir tras la orden adecuada la opción **--verify rutaArchivo.ino** y cada vez que queramos cargarlo en memoria (compilándolo en ese momento también), bastará con escribir la opción **--upload rutaArchivo.ino**. Hasta que no cambiemos el modelo de placa y/o el puerto serie utilizado, los comandos de compilación y carga usarán los valores de **--board** y **--port** configurados inicialmente. Así pues, si nuestro computador está funcionando sobre Linux y queremos, por ejemplo, compilar y además cargar en una placa Arduino UNO un sketch llamado "proj1.ino" (ubicado en la carpeta "/home/user/Arduino/proj1"), solo necesitaremos ejecutar el comando **./arduino --board arduino:avr:uno --port /dev/ttyACM0 --upload /home/user/Arduino/proj1/proj1.ino**. Si, a partir de esta primera carga, modificáramos el código de ese sketch y lo quisieramos volver a (compilar y) cargar en el mismo tipo de placa, tan solo haría falta escribir cada vez el comando **./arduino --upload /home/user/Arduino/proj1/proj1.ino**.

Además de estas labores habituales de compilar y cargar sketches, el IDE Arduino nos permite realizar más tareas desde el terminal. Por ejemplo, podemos instalar librerías (siempre que se hallen disponibles a través del "Library Manager") mediante el comando: **./arduino --install-library nombreLibrería:versión, otraLibreria:versión ...**, donde indicar ":versión" es opcional (si se omite, se instalará la versión más moderna posible). Si una librería de las indicadas en este comando ya estuviera instalada, la librería existente solo será reemplazada si la versión de la nueva es diferente. También podemos instalar un "cores" (siempre que se hallen disponibles a través del "Boards Manager") mediante el comando **./arduino --install-boards fabricante:arq:version**, donde indicar ":versión" es opcional (si se omite, se instalará la versión más moderna posible). Si el "core" indicado en este comando ya estuviera instalado, el "core" existente solo será reemplazado si la versión del nuevo es diferente.

Otras opciones interesantes del IDE Arduino funcionando como comando de terminal son, por ejemplo, la opción `--preserve-temp-files` (para no borrar –entre otros– los ficheros ".hex" resultantes de la compilación del código realizada por `--verify` o `--upload`, ya que por defecto este tipo de ficheros son temporales), la opción `--verbose-build` (para aumentar el nivel de detalle en los mensajes de compilación) o la opción `--verbose-upload` (para aumentar el nivel de detalle en los mensajes de carga del sketch), entre otras. Para más información sobre otras opciones o maneras de usar el IDE Arduino en un terminal, recomiendo consultar <https://github.com/arduino/Arduino/blob/master/build/shared/manpage.adoc>.

USO DEL IDE "ARDUINO CREATE"

"Arduino Create" (<https://create.arduino.cc>) es un entorno de desarrollo, también soportado oficialmente por Arduino, que tiene la particularidad de funcionar dentro del navegador de forma completamente "online". Para empezar a utilizarlo primero deberemos crearnos una cuenta de usuario en <https://id.arduino.cc/auth/signup>. Tras realizar este paso, ya podremos iniciar sesión con ella en "Arduino Create" (y no solamente en ese entorno: a través de <https://id.arduino.cc/auth/login> podremos loguearnos con esa misma cuenta en los foros de Arduino –<http://forum.arduino.cc>– o en la tienda de Arduino –<http://store.arduino.cc>–, entre otros lugares!).

Una vez dentro de "Arduino Create", podremos utilizar su completo editor web para escribir todos nuestros códigos, pudiéndolos grabar "en la nube" gracias al espacio de almacenamiento proporcionado por este mismo entorno (el llamado "Arduino Cloud"). También podemos subir allí sketches que tengamos ya escritos en nuestro computador; de esta manera los tendremos accesibles desde cualquier ordenador del mundo con acceso a Internet. En los códigos desarrollados mediante ese editor web online podremos incluir todas las librerías oficiales y las de terceros que estén accesibles vía el "Library Manager"; asimismo, podremos añadir para cualquier "core" que sea oficial o bien que esté accesible vía el "Boards Manager".

Lo más interesante del entorno "Arduino Create", no obstante, es la posibilidad que ofrece de compilar cualquier sketch, así como la de cargar ese sketch compilado en la placa que tengamos conectada en ese momento en el puerto USB de nuestro computador. Para esto último, no obstante, deberemos tener instalado un programa específico (el "Arduino Create Agent", <https://github.com/arduino/arduino-create-agent>) el cual permite a un navegador poder manejar el canal serie de nuestro computador (o más específicamente, a una página web remota tal como es el propio

EL MUNDO GENUINO-ARDUINO

"Arduino Create"). Este será el único software relacionado con el entorno "Arduino Create" que deberíamos instalar en nuestro computador (eso sí, solamente si queremos realizar cargas de sketches a placas físicas; si no, ni siquiera eso).

"Arduino Create" ofrece muchas más opciones, como un cuidado conjunto de proyectos ya elaborados disponibles como referencia, una gran variedad de tutoriales paso a paso incluyendo detallados esquemas, diagramas de los circuitos involucrados e incluso vídeos explicativos (también da la posibilidad de subir uno mismo sus propios esquemas y diagramas para sus propios proyectos), un conjunto de servicios de terceros "online" (como Temboo, por ejemplo) completamente integrados dentro del entorno, la posibilidad de compartir proyectos con otros usuarios así como también la de participar en proyectos de los demás, etc., etc.

Los objetivos de "Arduino Create" son, en definitiva, dos: fomentar la colaboración y el intercambio de implementaciones de ideas entre la comunidad de usuarios y simplificar el desarrollo de un proyecto en su totalidad evitando que el usuario tenga que cambiar de entorno, de recursos y de herramientas a medida que vaya avanzando en su proceso. Esto último se consigue gracias a que toda la documentación y referencias de los proyectos, sus códigos, sus diagramas, los compiladores y placas empleados, etc., se encuentran ubicados dentro de "Arduino Create" de una forma centralizada, integral, clara y organizada.

OTROS IDEs ALTERNATIVOS

Hay bastante gente a la que el IDE oficial de Arduino no le acaba de convencer, por varios motivos: la falta de funcionalidades avanzadas (como el autocompletado de sentencias, por ejemplo) o la dependencia del lenguaje Java (hecho que implica tener instalado en nuestro computador más paquetes de los realmente necesarios para poder compilar y cargar nuestros sketches) son algunos. Otra razón es que muchos programadores ya están familiarizados con un IDE concreto y quieren seguir utilizándolo para sus desarrollos con Arduino. La consecuencia de esto es que actualmente existe una variedad de IDEs "alternativos" que aportan más características que el IDE original o simplemente que cambian la manera de trabajar. A continuación, se nombran algunos de estos IDEs (no es una lista exhaustiva!) por si el lector quiere buscar en ellos alguna característica que el IDE oficial no tiene:

UECIDE (<http://uecide.org>): IDE libre y multiplataforma que unifica las diferentes variantes que existen del IDE oficial de Arduino para diferentes placas derivadas o similares (como chipKIT, Launchpad, etc.) en un solo IDE común.

Mariamole (<http://dalpix.com/mariamole>): IDE libre y multiplataforma con diversas características interesantes, como la posibilidad de trabajar en diversos proyectos a la vez, modificar los temas de color en el resultado del código fuente, configurar más detalladamente las opciones de compilación, etc. Para funcionar necesita que el IDE oficial de Arduino también esté instalado.

CodeBlocks (<http://www.codeblocks.org>): es un IDE libre y multiplataforma para el desarrollo de aplicaciones escritas en lenguaje C y C++. No obstante, se pueden escribir programas directamente en lenguaje Arduino (y también cargarlos en la placa) mediante una versión modificada de este software, la llamada "CodeBlocks Arduino Edition", descargable desde la dirección <http://www.arduinoide.com/codeblocks>.

CuWire (<http://apla.github.io/cuwire>): en realidad se trata de un complemento ("plug-in") para usar el lenguaje Arduino dentro del editor de texto libre y multiplataforma llamado Brackets (<http://brackets.io>). Para funcionar necesita que el IDE oficial de Arduino también esté instalado.

Arduino-Eclipse (<http://eclipse.baeyens.it>): en realidad se trata de un complemento ("plug-in") para usar el lenguaje Arduino dentro del entorno de programación libre y multiplataforma llamado Eclipse edición C/C++ (<http://www.eclipse.org>).

Visualmicro (<http://visualmicro.codeplex.com>): en realidad se trata de un complemento ("plug-in") para usar el lenguaje Arduino dentro de los entornos de programación gratuitos (pero sólo para Windows) Visual Studio Community (<http://www.visualstudio.com>) de Microsoft y Atmel Studio (<http://www.atmel.com/tools/atmelstudio.aspx>) de Atmel.

EmbedXcode (<http://embedxcode.weebly.com>): en realidad se trata de un complemento ("plug-in") del entorno de programación oficial de Apple, llamado XCode (<https://developer.apple.com/xcode>). Permite trabajar con un IDE "todo en uno" mediante el cual se pueden programar de forma unificada diversas plataformas, como Arduino, chipKIT, MSP430 o Wiring, (aunque cada una mediante su lenguaje de programación propio).

Si no queremos, de todas formas, utilizar ninguno de los entornos de desarrollo compatibles anteriores, aún tenemos la posibilidad de escribir código Arduino con absolutamente cualquier otro IDE que elijamos. Para ello, deberemos

EL MUNDO GENUINO-ARDUINO

tener instalado tanto nuestro IDE favorito como el entorno oficial de Arduino, y seguir los pasos indicados a continuación. Así lograremos utilizar todas las funcionalidades para la edición de código que ofrezca nuestro IDE favorito y utilizar el entorno oficial solo para compilar y cargar.

1. Ejecutar el IDE Arduino y abrir el fichero ".ino" que deseemos editar.
2. Ir al cuadro de preferencias del IDE y activar la opción "Use external editor". Automáticamente, el editor de código aparecerá en color gris, indicando que está deshabilitado.
3. Ejecutar el IDE que deseemos, y abrir el mismo fichero ".ino". Realizar la edición necesaria.
4. Grabar los cambios en el editor utilizado. Es importante que el fichero ".ino" grabado mantenga el mismo nombre que el fichero ".ino" originalmente abierto en el IDE Arduino, porque si no este no se enterará.
5. Utilizar los botones de "Verify" y "Upload" del IDE Arduino de la forma habitual, cuando se considere oportuno.

Entornos "online"

Otra opción es emplear IDEs de tipo "online", del estilo de "Arduino Create". Por ejemplo, **Codebender** (<http://www.codebender.cc>) ofrece, al igual que el proyecto oficial de Arduino, un entorno de desarrollo que funciona completamente dentro del navegador sin necesidad de instalar nada (a no ser que se deseé interactuar con una placa conectada a nuestro computador; en ese caso sí se deberá instalar un "plugin" específico para el navegador). Incluye un completo editor de textos, un compilador, un vasto conjunto de librerías de terceros para disponer en nuestros proyectos, un cargador real de sketches... y además, previo registro gratuito, permite almacenar "en la nube" el conjunto de códigos realizados. La mayor ventaja que ofrece Codebender respecto a "Arduino Create" (su competidor natural) es que no está limitado solamente a placas de tipo Arduino sino que su funcionalidad está disponible también para ser empleada con otros tipos de hardware. Por otro lado, este proyecto es libre, por lo que uno se puede descargar de la web <https://github.com/codebendercc/bachelor> el software necesario para montar una plataforma Codebender propia.

Mención aparte merece el portal **123DCircuits** (<http://123d.circuits.io>) el cual ofrece un completo entorno online desde donde podremos diseñar circuitos electrónicos con tan solo arrastrar y soltar sobre el espacio de trabajo (representado por una breadboard) los diferentes componentes necesarios (representados por sus

respectivas ilustraciones). Lo interesante está en que uno de esos elementos puede ser una placa Arduino, la cual, por el hecho de pertenecer al diseño del proyecto, permite ser programada mediante un IDE online asociado a este. De esta forma, podemos simular su funcionamiento dentro del circuito diseñado y, por tanto, observar su comportamiento sin necesidad ni siquiera de disponer de una placa Arduino real. Además, este portal incorpora la posibilidad de dibujar (y exportar) los esquemas eléctricos del circuito correspondientes e incluso generar la eventual PCB resultante. Muy completo.

Entornos de programación gráfica

También conviene destacar la existencia de un tipo de entornos de desarrollo para Arduino un tanto "especiales", ya que están enfocados a la programación visual de sketches. Es decir: en vez de utilizar instrucciones escritas en un lenguaje abstracto, los códigos se construyen a partir del "acoplamiento" gráfico de diferentes bloques coloreados (a modo de piezas de puzzle) que representan acciones y estructuras de control. Su objetivo es facilitar a cualquier persona sin ninguna experiencia en programación (por ejemplo, los niños) la iniciación en el mundo de los microcontroladores, matando de esta forma dos pájaros de un tiro: la introducción a la programación y la introducción a la electrónica. Los más destacables son:

Scratch for Arduino –S4A– (<http://s4a.cat>): plug-in multiplataforma y libre del entorno visual de programación Scratch (<http://scratch.mit.edu>) que permite usar dicho entorno visual para programar e interactuar con placas Arduino. En otras palabras, S4A añade al entorno Scratch estándar un conjunto extra de bloques coloreados especializados en gestionar la interacción entre la placa Arduino y todo tipo de sensores, actuadores, etc. Hay que hacer notar, no obstante, que para que S4A funcione, previamente se habrá tenido que cargar un sketch específico en la memoria del microcontrolador de la placa a manejar (el cual será responsable de establecer comunicación –de tipo Firmata concretamente– con el exterior) y dicha placa, además, deberá estar en contacto permanente con el computador que esté ejecutando S4A. Otro proyecto similar (que también obliga a tener la placa conectada a nuestro computador y haber cargado en ella un sketch concreto haciendo uso de Firmata) es "**Scratch Arduino Extension**" (<https://khanning.github.io/scratch-arduino-extension>), el cual se basa, en este caso, en el sistema general de extensiones ScratchX (<http://scratchx.org>) para dotar al entorno Scratch estándar de la posibilidad de interactuar con hardware real (notar que para que ScratchX funcione deberemos instalar antes un "plugin" en nuestro navegador tal como se indica en https://scratch.mit.edu/info/ext_download).

EL MUNDO GENUINO-ARDUINO

Finalmente, otro proyecto (más modesto) con los mismos objetivos que los dos anteriores es **SuperEasy-A4S** (<http://thomaspreece.com/resources.php>).

Snap for Arduino (<http://s4a.cat/snap>): plug-in multiplataforma y libre del entorno visual de programación Snap (<http://snap.berkeley.edu>) que permite usar dicho entorno visual para programar e interactuar con placas Arduino. En otras palabras, es un complemento que añade al entorno Snap estándar un conjunto extra de bloques coloreados especializados en gestionar la interacción entre la placa Arduino y todo tipo de sensores, actuadores, etc. Hay que hacer notar, no obstante, que para que "Snap for Arduino" funcione, previamente se habrá tenido que cargar un sketch específico en la memoria del microcontrolador de la placa a manejar (el cual será responsable de establecer comunicación –de tipo Firmata concretamente– con el exterior) y dicha placa, además, deberá estar en contacto permanente con el computador que esté ejecutando "Snap for Arduino". En su última versión, no obstante, es capaz de generar código Arduino estándar a partir del diseño de bloques realizado, por lo que ese código puede ser cargado en la memoria del microcontrolador de la placa como un sketch cualquiera más. Otro proyecto similar es **S2A_fm** (https://github.com/MrYsLab/s2a_fm).

NOTA: Lo más interesante que aporta Snap respecto Scratch es la posibilidad de poder definir y crear uno mismo los bloques coloreados que se deseen si no están disponibles por defecto; de esta manera, Snap ofrece una manera sencilla de extenderse a si mismo y ampliar el lenguaje básico para lograr así derivaciones especializadas muy interesantes, como por ejemplo (por citar una) BeetleBlocks (<http://beetleblocks.com>), enfocada en el diseño de imágenes en 3D.

BlocklyDuino (<https://github.com/BlocklyDuino/BlocklyDuino>): editor visual y libre basado en el navegador (y por tanto, multiplataforma). Está desarrollado mediante Blockly (<https://developers.google.com/blockly>), un entorno de programación similar en espíritu a Snap (permite, como él, crear nuevos bloques coloreados personalizados) pero enfocado únicamente al navegador como entorno de trabajo (es decir, no ofrece, como Scratch/Snap, la opción de ser utilizado en forma de IDE instalable). Por otro lado, y al igual que Snap, es capaz de generar código Arduino estándar a partir del diseño de bloques realizado, por lo que ese código puede ser cargado en la memoria del microcontrolador de la placa como un sketch cualquiera más. Otro proyecto similar es **Bitbloq** (<http://bitbloq.bq.com>).

Minibloq (<http://blog.minibloq.org>): entorno de desarrollo visual muy similar a Scratch/Snap, libre pero solo para Windows. Tiene la ventaja, no obstante, de poderse utilizar en otras arquitecturas hardware además de las

arquitecturas de las placas Arduino. Al igual que las herramientas mencionadas en el párrafo anterior, es capaz de generar el código Arduino interno equivalente a los diferentes bloques visuales utilizados, permitiendo así, en un momento puntual, personalizar los sketches más directamente.

Ardublock (<http://blog.ardublock.com>): entorno de desarrollo visual para Arduino similar al anterior. No obstante, al estar programado en Java, es multiplataforma (aunque para funcionar necesita que el IDE oficial de Arduino esté previamente instalado).

MÁS ALLÁ DEL LENGUAJE ARDUINO: EL LENGUAJE C/C++

A lo largo de los párrafos anteriores hemos repetido varias veces la palabra "compilación", pero no hemos explicado su significado hasta ahora. "Compilar" significa convertir un código escrito en el IDE (utilizando el lenguaje Arduino, y por tanto, entendible fácilmente por los seres humanos) en el programa realmente ejecutable por el microcontrolador, que no es más que un inmenso conjunto de bits (es decir, 1s y 0s) tan solo entendible por él.

Es decir, nosotros escribimos en el IDE las instrucciones usando el sencillo lenguaje de programación Arduino y posteriormente, mediante la compilación, transformamos ese sketch en instrucciones "digeribles" para el microcontrolador (en lo que viene a llamarse "código máquina" o "código binario"). Este código máquina en esencia no es más que un conjunto de impulsos eléctricos (1s –pasa corriente– y 0s –no pasa corriente–), que es lo único que realmente saben procesar los circuitos electrónicos. Así pues, un código máquina (ficticio) resultante de la compilación de un sketch Arduino cualquiera podría ser similar a esto: 10010111010101011011...

Es evidente que es imposible escribir un programa directamente en código máquina: por eso existen los compiladores. Pero, además, estas herramientas ofrecen otra ventaja: hay que saber que el código máquina válido para un tipo de microcontrolador no lo es para otro, debido a su diferente construcción electrónica interna. Por lo tanto, un mismo programa se tendría que codificar en diferentes códigos máquina para diferentes modelos de microcontrolador. Sin embargo, si disponemos de compiladores específicos para cada uno de estos modelos, a partir de un mismo código fuente podemos obtener diferentes códigos máquina utilizando en cada caso el compilador respectivo. Esto es una gran ventaja, porque nos permite no tener que reescribir para distintas placas un mismo código ya funcional, evitándonos así cometer posibles errores en las reescrituras y, sobre todo, aportando una gran flexibilidad y escalabilidad a nuestros proyectos. Concretamente, el entorno de

EL MUNDO GENUINO-ARDUINO

programación oficial de Arduino incluye por defecto un compilador específico para generar código binario compatible con las placas Arduino de tipo AVR y otro diferente para las placas con microcontroladores de tipo ARM, eligiendo automáticamente el compilador adecuado dependiendo de la placa concreta que esté indicada en el menú "Tools->Board" en ese momento.

Sin embargo, ninguno de estos dos compiladores incluidos en el IDE de Arduino compilan código escrito en lenguaje Arduino a código binario AVR/ARM, sino que compilan código escrito en lenguaje C/C++ a código binario AVR/ARM. ¿Por qué? Porque realmente, el lenguaje Arduino no es un lenguaje en sentido estricto: es simplemente un conjunto de instrucciones C y C++ "camufladas", diseñadas para simplificar el desarrollo de programas para microcontroladores soportados por el ecosistema Arduino. Es decir, cuando estamos escribiendo nuestro sketch en "lenguaje Arduino", sin saberlo en realidad estamos programando en una versión simplificada del lenguaje C/C++.

El lenguaje C y su "pariente" C++ son dos de los lenguajes más importantes y extendidos del mundo por varias razones: porque son lenguajes muy potentes y a la vez ligeros y flexibles, porque poseen un conjunto amplísimo de librerías que los dotan de funcionalidad que otros lenguajes no ofrecen, porque los programas escritos y compilados en estos lenguajes son tremadamente eficientes y rápidos, y porque existen compiladores para prácticamente cualquier tipo de hardware (con lo que hoy en día podemos ver multitud de software escrito con estos lenguajes ejecutándose en una gran variedad de máquinas).

No obstante, tal vez para una persona que se inicia en el mundo de la programación, los lenguajes C/C++ no son demasiado amigables. En otras palabras: son lenguajes relativamente difíciles de aprender y dominar. Por eso existe el lenguaje Arduino: para que una persona sin apenas conocimientos de desarrollo de aplicaciones pueda escribir rápidamente un sketch Arduino funcional sin tener que aprender todo un lenguaje de programación completo pero complejo como es C o C++. Es decir: el lenguaje Arduino hace de "máscara" del lenguaje C/C++, ocultando gran cantidad de detalles superfluos para el entusiasta de los proyectos electrónicos y facilitando así el uso de la plataforma Arduino en conjunto. El "precio" a pagar por ganar esta facilidad en la programación de sketches es que tan solo tenemos un subconjunto de toda la funcionalidad que puede llegar a ofrecer el lenguaje C/C++. Afortunadamente, para la gran mayoría de proyectos no es necesario ir más allá de lo que ya nos ofrece el lenguaje Arduino y su IDE oficial, y por supuesto, todos los ejemplos de este libro están programados utilizando solo las funcionalidades que ofrece el lenguaje Arduino en exclusiva.

Herramientas de compilación C/C++ y carga incluidas en el IDE

Si el lector desea conocer en detalle todo el proceso interno de transformación del sketch originalmente escrito en lenguaje Arduino a su versión en lenguaje C/C++, su posterior compilación a código binario AVR/ARM y su carga final al microcontrolador de la placa, le recomiendo que consulte el enlace oficial <https://github.com/arduino/Arduino/wiki/Build-Process> porque allí podrá obtener toda la información que necesite sobre este asunto. De todas formas, para su comodidad a continuación resumimos los pasos más importantes de dicho proceso.

El desarrollo típico de un sketch Arduino consta de los siguientes pasos:

- Escritura del código fuente en lenguaje "Arduino" (aunque también sería perfectamente posible escribir nuestro sketch directamente empleando el lenguaje C/C++ "puro").
- Transformación del código fuente "Arduino" escrito en el paso anterior a código C/C++ "puro" (a este proceso se le llama "preprocesado"), incrustando en él también el código C/C++ propio de las librerías empleadas en ese sketch en particular (a este proceso se le llama "enlazado") y su subsiguiente compilación total. Todo esto se realiza mediante el botón "Verify".
- Si el paso anterior no ha devuelto errores, carga del código binario resultante en la memoria del microcontrolador de la placa Arduino utilizada. Esto se realiza mediante el botón "Upload".

Cada una de los pasos anteriores es realizado internamente por una parte separada del IDE. De hecho, estas partes son tan independientes entre sí que, en la práctica, el IDE Arduino lo único que hace es utilizar diferentes herramientas autónomas para realizar cada uno de estos pasos. Esto es algo que nosotros podríamos hacer también de forma manual invocando (normalmente dentro de un intérprete de comandos) a cada una de estas herramientas por separado, pero, como es lógico, no sería un procedimiento tan cómodo como el ofrecido por el IDE oficial, consistente en simplemente escribir el código y hacer clic en los botones "Verify" y "Upload".

Herramientas invocadas mediante el botón "Verify"

La primera herramienta interna que el IDE Arduino pone en marcha (tras haber "desmaquillado" internamente el código Arduino escrito por el desarrollador para convertirlo en código C/C++ "puro") es un compilador cruzado C/C++, ya sea el específico para placas AVR o el específico para ARM (según el tipo de placa que

EL MUNDO GENUINO-ARDUINO

estemos usando). Que un compilador sea "cruzado" significa simplemente que genera código binario ejecutable en una plataforma diferente –Arduino– de la plataforma donde está instalado –computador-. Como ya sabemos, dicha herramienta (o mejor dicho, el conjunto de herramientas, generalmente ejecutables en forma de distintos comandos de terminal) es la que se encargará de obtener, a partir del código fuente del sketch (previamente escrito y guardado en un fichero de nuestro computador) y el de las eventuales librerías importadas por este, un determinado código binario, generalmente en forma de un nuevo fichero. El código binario generado por un compilador para la arquitectura AVR tiene un formato llamado "Intel .hex" (o simplemente, ".hex") que es compatible, lógicamente, con los microcontroladores AVR de Atmel; el código binario generado por un compilador para la arquitectura ARM puede tener, en cambio, muchos formatos diferentes, pero el utilizado en los microcontroladores de Atmel (los llamados "SAM") es uno en concreto llamado genéricamente ".bin".

NOTA: En el caso de utilizar el IDE oficial, el código binario generado (ya sea ".hex" o ".bin") solo lo guarda de forma temporal hasta que es cargado en la memoria del microcontrolador, pero nada impide que dicho código pueda ser almacenado permanentemente en un fichero de nuestro computador (que es lo que permite, de hecho, la opción "Sketch->Export compiled binary" del propio IDE).

El compilador C/C++ para placas AVR que utiliza el IDE Arduino es un programa llamado "gcc-avr". Si quisiéramos utilizarlo de forma independiente, podríamos descargarlo directamente de su web oficial (<http://gcc.gnu.org>) o, más fácil, en el caso de usar Ubuntu, obtener el paquete homónimo en su "Centro de software" (en el caso de Fedora, deberíamos descargar dos paquetes con los nombres de "avr-gcc" y "avr-gcc-c++"). De todas formas, instalar solamente "gcc-avr" en nuestro computador no sería suficiente: para realizar las compilaciones correctamente se necesita tener instalado además un conjunto de librerías básicas estándar del lenguaje C para la plataforma AVR llamado "avr-libc" (su web oficial es <http://www.nongnu.org/avr-libc> y Atmel ofrece una detallada documentación en <http://www.atmel.com/webdoc/AVRLibcReferenceManual/index.html>; tanto en Ubuntu como en Fedora estas librerías se encuentran disponibles dentro de un paquete homónimo).

NOTA: Además de "avr-libc", también sería necesario tener instalado el conjunto de utilidades llamadas genéricamente "binutils" (el paquete correspondiente en Ubuntu se llama "binutils-avr" y en Fedora "avr-binutils"), las cuales realizan tareas de ensamblaje y enlace entre las librerías "avr-libc" y otras librerías necesarias para lograr una correcta compilación. Su web oficial es: <http://www.gnu.org/software/binutils>

El compilador C/C++ para placas ARM que viene integrado en el IDE Arduino es uno específico para arquitecturas de tipo Cortex-M0/M0+/M3/M4/M7 y Cortex-

R4/R5/R7 (entre las cuales, precisamente, se encuentran las placas Due y Zero). Si lo quisieramos utilizar de forma independiente, podríamos descargar todo lo necesario de su web oficial (<https://launchpad.net/gcc-arm-embedded>) o bien, en el caso de usar Ubuntu, de forma equivalente podríamos obtener el paquete "gcc-arm-none-eabi" de su "Centro de software" (también necesitaríamos los paquetes "libnewlib-arm-none-eabi", "libstdc++-arm-none-eabi-newlib" y "binutils-arm-none-eabi"; en Fedora los paquetes necesarios se llaman "arm-none-eabi-gcc-cs", "arm-none-eabi-gcc-cs-c++", "arm-none-eabi-binutils-cs" y "arm-none-eabi-newlib").

NOTA: La razón del nombre tan "extraño" de los paquetes relacionados con el compilador para ARM y utilidades y librerías asociadas es la siguiente: "none" indica que la plataforma sobre la que se ejecutará el código binario compilado es un microcontrolador "a pelo", sin ninguna capa intermedia de software (es decir, en comunicación directa con el hardware); esto es importante especificarlo porque los compiladores ARM también son capaces de generar código binario preparado para ejecutarse sobre un sistema operativo, siendo en este caso un código binario muy diferente al aprovechar las funcionalidades que el software intermedio ofrece (por eso podemos encontrarnos con paquetes como "gcc-arm-linux-eabi", por ejemplo). La terminación "eabi", por su parte, proviene de "Embedded ABI" y es una forma de decir que el código binario generado está diseñado para funcionar en dispositivos de propósito específico (como son los microcontroladores, con o sin sistema operativo integrado) en contraposición a dispositivos de propósito general (como son los microprocesadores de las computadoras, por ejemplo).

Herramientas invocadas mediante el botón "Upload"

La herramienta "avrdude" (utilizable también directamente a través de un intérprete de comandos) es la que utiliza el IDE internamente para cargar desde un computador los ficheros ".hex" en la memoria de los microcontroladores de tipo AVR (vía USB con la ayuda del "bootloader" que tenga incorporado ese micro, o bien directamente a través de un programador ISP). Su web oficial es: <http://www.nongnu.org/avrdude>. Tanto en Ubuntu como en Fedora, "avrdude" se encuentra disponible de forma independiente dentro de un paquete homónimo.

Para cargar ficheros ".bin" en las placas de tipo ARM (Due y Zero) el IDE utiliza otro programa distinto, mantenido por el propio equipo de Arduino y disponible en <https://github.com/shumatech/BOSSA/tree/arduino>. Este programa (llamado "Bossa" pero cuyo comando principal es "bossac") deriva de otra aplicación homónima pero más genérica (<http://www.shumatech.com/web/products/bossa>) que pretende ser –tal como indica su nombre, "Basic Open Source SAM-BA"– una alternativa libre al programa "SAM Boot Assistant", más conocido como SAM-BA (<http://www.atmel.com/tools/ATMELSAM-BAIN-SYSTEMPROGRAMMER.aspx>), que es la aplicación oficial (multiplataforma y gratuita, pero no libre) ofrecida por Atmel para cargar desde un computador los ficheros ".bin" en la mayoría de sus

EL MUNDO GENUINO-ARDUINO

microcontroladores de tipo SAM (generalmente vía USB gracias al bootloader que llevan incorporado, el cual también se llama SAM-BA). En principio es indiferente que usemos el software SAM-BA, el "Bossa" original o el "Bossa" adaptado por Arduino para realizar las cargas de código binario a las placas Due y Zero, pero por compatibilidad se recomienda utilizar esta última herramienta.

LENGUAJE ARDUINO



MI PRIMER SKETCH ARDUINO

Conecta la placa Arduino a tu computador y ejecuta el IDE oficial. Selecciona (si no lo está ya) el tipo de placa y el puerto USB adecuados (en el menú *Tools->Board* y *Tools->Serial port*, respectivamente). Crea un nuevo sketch con el siguiente contenido:

Ejemplo 4.1

```
/*Declaración e inicialización de una  
variable global llamada 'mivariable' */  
int mivariable=555;  
void setup() {  
    Serial.begin(9600);  
}  
void loop() {  
    Serial.println(mivariable);  
    mivariable=mivariable+1;  
}
```

Pulsa en el botón "Verify" y seguidamente en el botón "Upload". No deberías de observar ningún error en la consola de mensajes. Abre ahora el "Serial monitor" y verás que allí van apareciendo en tiempo real muchos números uno tras otro,

EL MUNDO GENUINO-ARDUINO

empezando por el 555 y siguiendo por el 556, 557, 558, 559, etc., aumentando sin parar. ¿Por qué? ¿Qué significa este texto (este código) que hemos introducido en la memoria del microcontrolador de la placa?

ESTRUCTURA GENERAL DE UN SKETCH

Un programa diseñado para ejecutarse sobre un Arduino (un "sketch") siempre se compone de tres secciones:

La sección de declaraciones de variables globales: ubicada directamente al principio del sketch.

La sección llamada *void setup()*: delimitada por llaves de apertura y cierre.

La sección llamada *void loop()*: delimitada por llaves de apertura y cierre.

La primera sección del sketch (que no tiene ningún tipo de símbolo delimitador de inicio o de final) está reservada para escribir, tal como su nombre indica, las diferentes declaraciones de variables que necesitemos. En un apartado posterior explicaremos ampliamente qué significa todo esto.

En el interior de las otras dos secciones (es decir, dentro de sus llaves) deberemos escribir las instrucciones que deseemos ejecutar en nuestra placa, teniendo en cuenta lo siguiente:

Las instrucciones escritas dentro de la sección *void setup()* se ejecutan una única vez, en el momento de encender (o resetear) la placa Arduino.

Las instrucciones escritas dentro de la sección *void loop()* se ejecutan justo después de las de la sección *void setup()* infinitas veces hasta que la placa se apague (o se resetee). Es decir, el contenido de *void loop()* se ejecuta desde la 1^a instrucción hasta la última, para seguidamente volver a ejecutarse desde la 1^a instrucción hasta la última, para seguidamente ejecutarse desde la 1^a instrucción hasta la última, y así una y otra vez.

Por tanto, las instrucciones escritas en la sección *void setup()* normalmente sirven para realizar ciertas preconfiguraciones iniciales y las instrucciones del interior de *void loop()* son, de hecho, el programa en sí que está funcionando continuamente.

En el caso concreto del ejemplo 4.1, vemos que en la zona de declaraciones de variables globales hay una sola línea (`int mivariable=555;`) que dentro de `void setup()` se ejecuta una sola instrucción (`Serial.begin(9600);`) y que dentro de `void loop()` se realiza la ejecución continua y repetida (hasta que la alimentación de la placa se interrumpa) de dos instrucciones una tras otra: `Serial.println(mivariable);` y `mivariable=mivariable+1;` . Sobre el significado y sintaxis de todas estas líneas hablaremos a continuación.

Sobre las mayúsculas, tabulaciones y los punto y coma

Conviene aclarar ya pequeños detalles que deberemos tener en cuenta a la hora de escribir nuestros sketches para evitarnos muchos dolores de cabeza. Por ejemplo, es necesario saber que el lenguaje Arduino es "case-sensitive". Esto quiere decir que es totalmente diferente escribir una letra en mayúscula que en minúscula. Dicho de otra forma: para el lenguaje Arduino "Hola" y "hOLA" son dos palabras distintas. Esto tiene una implicación muy importante: no es lo mismo escribir por ejemplo `Serial.begin(9600);` que `serial.begin(9600);`. En el primer caso la instrucción estaría correctamente escrita, pero en el segundo, en el momento de compilar el código el IDE se quejaría porque para él la palabra `serial` (con "s" minúscula) no tiene ningún sentido. Así que hay que vigilar mucho con respetar esta distinción en los códigos que escribamos.

Otro detalle: las tabulaciones de las instrucciones contenidas dentro de las secciones `void setup()` y `void loop()` del sketch del ejemplo 4.1 no son en absoluto necesarias para que la compilación del sketch se produzca con éxito. Simplemente son una manera de escribir el código de forma ordenada, clara y cómoda para el programador, facilitándole la tarea de leer código ya escrito y mantener una cierta estructura a la hora de escribirlo. En los próximos ejemplos de este libro se irá viendo mejor su utilidad.

Otro detalle: todas las instrucciones (incluyendo también las declaraciones de variables) acaban con un punto y coma. Es indispensable añadir siempre este signo para no tener errores de compilación, ya que el compilador necesita localizarlo para poder detectar el final de cada instrucción escrita en el sketch. Si se olvida, se mostrará un texto de error que puede ser obvio ("falta un punto y coma") o no. Si el texto del error es muy oscuro o sin lógica, es buena idea comprobar que la causa no sea la falta de un punto y coma en las líneas justamente anteriores a la marcada por el compilador como causante del problema.

COMENTARIOS

La primera línea del sketch del ejemplo 4.1 contiene un comentario (concretamente, son las dos primeras líneas: desde los símbolos `/*` hasta los símbolos `*/`). Un "comentario" es un texto escrito intercalado con el código del sketch que se utiliza para informar sobre cómo funciona ese código a la persona que en algún momento lo esté leyendo. Es decir, los comentarios son texto de ayuda para los seres humanos que explican el código asociado y ayudan a entenderlo y recordar su función. Los comentarios son completamente ignorados y desechados por el compilador, por lo que no forman parte nunca del código binario que ejecuta el microcontrolador (así que no ocupan espacio en su memoria).

Los comentarios pueden aparecer dentro del código de diferentes formas:

Comentarios compuestos por una línea entera (o parte de ella): para añadirlos deberemos escribir dos barras (`//`) al principio de cada línea que queramos convertir en comentario. También podemos comentar solamente una parte de la línea, si escribimos las barras en otro punto que no sea el principio de esta; de esta manera solamente estaremos comentando lo que aparece detrás de las barras hasta el final de la línea, pero lo anterior no.

Comentarios compuestos por un bloque de varias líneas seguidas: para añadirlos deberemos escribir una barra seguida de un asterisco (`/*`) al principio del bloque de texto que queramos convertir en comentario, y un asterisco seguido de una barra (`*/`) al final de dicho bloque. Todos los caracteres y líneas ubicados entre estas dos marcas de inicio y final serán tratadas automáticamente como comentarios. Hay que tener en cuenta, por otro lado, que comentarios unilineales se pueden escribir dentro de un comentario multilineal, pero uno multilineal dentro de otro no. Este es el tipo de comentario escrito en el sketch del ejemplo 4.1.

Una práctica bastante habitual en programación es comentar en algún momento una (o más) partes del código. De esta forma, se "borran" esas partes (es decir, se ignoran, y por tanto, ni se compilan ni se ejecutan) sin borrarlas realmente. Normalmente, esto se hace para localizar posibles errores en el código observando el comportamiento del programa con esas determinadas líneas comentadas. A lo largo de los ejemplos de este libro se irá viendo su utilidad.

VARIABLES

La primera línea del sketch del ejemplo 4.1 consiste en declarar una variable global de tipo *int* llamada *mivariable*, e inicializarla con el valor de 555. Expliquemos todo esto.

Una variable es un elemento de nuestro sketch que actúa como un pequeño "cajoncito" (identificado por un nombre elegido por nosotros) que guarda un determinado contenido. Ese contenido (lo que se llama el valor de la variable) se podrá modificar en cualquier momento de la ejecución del sketch: de ahí el nombre de "variable". La importancia de las variables es inmensa, ya que todos los sketches hacen uso de ellas para alojar los valores que necesitan para funcionar.

El valor de una variable puede haberse obtenido de diversas maneras: puede haber sido asignado literalmente (como el del ejemplo 4.1, donde nada más empezar asignamos explícitamente a la variable llamada *mivariable* el valor 555), pero también puede ser el dato obtenido por un sensor, o el resultado de un cálculo, etc. Provenga de donde provenga inicialmente, ese valor podrá ser siempre cambiado en cualquier instante posterior de la ejecución del sketch, si así lo deseamos.

Declaración e inicialización de una variable

Antes de poder empezar a utilizar cualquier variable en nuestro sketch; no obstante, primero deberemos crearla. Al hecho de crear una variable se le suele llamar "declarar una variable". En el lenguaje Arduino, cuando se declara una variable es imprescindible además especificar su tipo. El tipo de una variable lo elegiremos según el tipo de datos (números enteros, números decimales, cadena de caracteres, etc.) que queramos almacenar en esa variable. Es decir: cada variable puede guardar solamente valores de un determinado tipo, por lo que deberemos decidir en su declaración qué tipo de variable es la que nos interesa más según el tipo de datos que preveamos almacenar. Asignar un valor a una variable que sea de un tipo diferente al previsto para esta provoca un error del sketch.

Los tipos posibles del lenguaje Arduino los detallaremos en los párrafos siguientes, pero ya se puede saber que la sintaxis general de una declaración es siempre una línea escrita así: *tipoVariable nombreVariable;*. En el caso concreto de querer declarar varias variables del mismo tipo, en vez de escribir una declaración por separado para cada una de ellas, es posible declararlas todas en una misma línea, genéricamente así: *tipoVariable nombreVariable1, nombreVariable2;*.

EL MUNDO GENUINO-ARDUINO

Opcionalmente, a la vez que se declara la variable, se le puede establecer un valor inicial: a esto se le llama "inicializar una variable". Inicializar una variable cuando se declara no es obligatorio, pero sí muy recomendable. En el ejemplo 4.1, declaramos la variable llamada *mivariable* de tipo *int* (que es un tipo de dato de entre los varios existentes pensados para guardar números enteros) y además, también la inicializamos con el valor inicial de 555. De ahí ya se puede deducir que la sintaxis general de una inicialización es siempre una línea escrita así: *tipoVariable nombreVariable = valorInicialVariable;*

Uno podría preguntarse por qué es conveniente usar variables inicializadas en vez de escribir su valor directamente allí donde sea necesario. Por ejemplo, en el caso del código del ejemplo 4.1, hemos asignado el valor de 555 a *mivariable* cuando lo podríamos haber escrito directamente dentro de la instrucción *Serial.println()* así: *Serial.println(555);*. La razón principal es porque trabajar con variables nos facilita mucho la comprensión y el mantenimiento de nuestros programas: si ese valor aparece escrito en multitud de líneas diferentes de nuestro código y ha de ser cambiado, utilizando una variable inicializada con ese valor tan solo deberíamos cambiar la inicialización de la variable y automáticamente todas las veces donde apareciera esa variable dentro de nuestro código tendría este nuevo valor; si escribiéramos directamente ese valor en cada línea, tendríamos que irlo modificando línea por línea, con la consecuente pérdida de tiempo y la posibilidad de cometer errores.

Por otro lado, a la hora de declarar las variables, se recomienda dar a nuestras variables nombres descriptivos para hacer el código de nuestro sketch más legible. Por ejemplo, nombres como *sensorDistancia* o *botonEncendido* ayudarán a entender mejor lo que esas variables representan. Para nombrar una variable se puede utilizar cualquier palabra que queramos, siempre que esta no sea ya una palabra reservada del lenguaje Arduino (como el nombre de una instrucción, etc.) y que no empiece por un dígito.

Asignación de valores a una variable

¿Qué pasa si una variable se declara pero no se inicializa? Pues que tendrá un determinado valor por defecto dependiendo del tipo que dato que esté previsto guardar en ella. Por ejemplo, si fuera una variable diseñada para contener un dato de tipo numérico entero, su valor predeterminado es 0; si alojara un dato de tipo numérico decimal, ese valor es 0.00; si alojara un carácter o un conjunto seguido de ellos (lo que se llama una "cadena"), entonces es la cadena vacía (""), etc. Y ese valor por defecto que tenga la variable lo mantendrá hasta que se le asigne un valor

diferente en algún momento de la ejecución de nuestro sketch. La sintaxis general para asignar un nuevo valor a una variable (ya sea porque no se ha inicializado, o sobre todo porque se desea sobrescribir un valor anterior por otro nuevo), es: `nombreVariable = nuevoValor;`.

Por ejemplo, si ya hubiéramos declarado una variable de tipo entero con el nombre de *mivariable*, para asignarle un nuevo valor numérico (por ejemplo, 567) podría escribirse una línea tal como `mivariable=567`; ya sea en el interior de la sección `void setup()`—si queremos que esa asignación se realice solamente una vez, al principio de la ejecución de nuestro sketch (a modo de inicialización)—, o bien en el interior de la sección `void loop()`—si queremos que esa asignación se repita infinitas veces—. Importante fijarse, en cualquier caso, que la línea de asignación se lee "de derecha a izquierda": es decir, el valor *nuevoValor* se asigna a la variable *nombreVariable*.

Las asignaciones de valores a variables pueden ser muy variadas: no siempre son valores directos como en el ejemplo 4.1. Un caso bastante habitual es asignar a una variable un valor que depende del valor de otra. Por ejemplo, si suponemos que tenemos una variable llamada *y* y otra llamada *x*, podríamos escribir lo siguiente: `y = x + 10;`. La sentencia anterior se ha de leer así: al valor que tenga actualmente la variable *x* se le suma 10 y el resultado se asigna a la variable *y* (es decir, que si por ejemplo *x* tiene un valor de 5, *y* tendrá un valor de 15).

Incluso nos podemos encontrar con asignaciones del tipo `y = y + 1;` (como de hecho ocurre con el sketch del ejemplo 4.1). La clave aquí está en entender que el símbolo "`=`" no es el de la igualdad matemática (que ya estudiaremos) sino el de la asignación. Por tanto, una línea como la anterior lo que hace es sumar una unidad al valor actual de la variable *y* (se entiende que es de tipo numérico) para seguidamente asignar este nuevo valor resultante otra vez a la variable *y*, sobrescribiendo el que tenía anteriormente. Es decir, que si inicialmente *y* vale 45 (por ejemplo), después de ejecutarse la asignación `y = y + 1;` valdrá 46.

Ámbito de una variable

Otro concepto importante en relación con las variables es el de ámbito de una variable. En este sentido, una variable puede ser "global" o "local". Que sea de un ámbito o de otro depende de en qué lugar de nuestro sketch se declare la variable:

Para que una variable sea global se ha de declarar al principio de nuestro sketch; es decir, antes (y fuera) de las secciones `void setup()` y `void loop()`. De hecho, al hablar de la estructura de un sketch ya habíamos mencionado la

EL MUNDO GENUINO-ARDUINO

existencia de esta sección de declaraciones de variables globales. Una variable global es aquella que puede ser utilizada y manipulada desde cualquier punto del sketch. Es decir, todas las instrucciones de nuestro programa, sin importar dentro de qué sección estén escritas (`void setup()`, `void loop()` u otras que puedan existir) pueden consultar y también cambiar el valor de dicha variable.

Para que una variable sea local se ha de declarar en el interior de alguna de las secciones de nuestro sketch (es decir, dentro de `void setup()` o de `void loop()` o de otras que puedan existir –y que ya iremos conociendo–, como son las funciones propias, los bloques condicionales o los bucles). Una variable local es aquella que solo puede ser utilizada y manipulada por las instrucciones escritas dentro de la misma sección donde se ha declarado. Este tipo de variables es útil en sketches largos y complejos para asegurarse de que solo una sección tiene acceso a sus propias variables, ya que esto evita posibles errores cuando una sección modifica inadvertidamente variables utilizadas por otra sección.

Tipos posibles de una variable

Los tipos de variables que el lenguaje Arduino admite pueden ser clasificados en dos categorías, según si son tipos simples o complejos. Ambos se describen a continuación.

Tipos simples

El tipo `boolean`: las variables de este tipo solo pueden tener dos valores: cierto o falso. Se utilizan para almacenar un estado de entre esos dos posibles, y así hacer que el sketch reaccione según detecte en ellas uno u otro. Por ejemplo, las variables booleanas se pueden usar para comprobar si se han recibido datos de un sensor (cierto) o no (falso), para comprobar si algún actuador está disponible (cierto) o no (falso), para comprobar si el valor de otra variable diferente cumple una determinada condición como por ejemplo ser mayor que un número concreto (cierto) o no (falso). El valor guardado en una variable booleana ocupa siempre un byte de memoria.

Para asignar explícitamente a una variable de tipo `boolean` el valor de cierto, se puede utilizar la palabra especial `true` (sin comillas) o bien el valor `1` (sin comillas), y para asignarle el valor de falso se puede utilizar la palabra especial `false` (sin comillas) o bien el valor `0` (sin comillas). Es decir, si en nuestro sketch de ejemplo la variable `mivariable` hubiera sido de tipo `boolean` en vez de `int`, para asignarle el valor

CAPÍTULO 4: LENGUAJE ARDUINO

de *true* tendríamos que haber escrito una línea similar a `boolean mivariable=true;` o bien `boolean mivariable=1;` (en realidad, una variable booleana con un valor cualquiera diferente de 0 ya se interpreta que tiene un valor cierto: no tiene por qué ser el valor 1 concretamente).

El tipo *char*: el valor que puede tener una variable de este tipo es un número entero entre -128 y 127 (esto es debido a que cada valor de tipo *char* solo ocupa 8 bits en memoria, y por tanto, tal como nos indica la fórmula de las "variaciones con repetición" vista en el capítulo anterior, en este caso solo existen $2^8=256$ valores diferentes posibles). No obstante, lo más interesante del tipo *char* (su nombre ya lo indica, de hecho) es que el valor de una variable de este tipo –que no deja de ser un valor numérico, recordemos– puede ser asignado y manipulado dentro de nuestros sketches como si fuera un carácter (es decir, una letra, un dígito, un signo de puntuación...). Expliquemos esto.

Ya hemos dicho que los dispositivos electrónicos digitales solo son capaces de trabajar con números (binarios). Esto significa que no entienden ni reconocen los símbolos presentes en los distintos lenguajes humanos. Por tanto, para un dispositivo electrónico digital, ningún carácter (como por ejemplo 'a', '*', '6', '€'...) significa nada. Así pues, para que estos dispositivos puedan tratar con símbolos lingüísticos, debe existir un mecanismo de "traducción" que transforme esos caracteres (provenientes del exterior humano) a números binarios (y así conseguir que el dispositivo los pueda almacenar y procesar correctamente), y viceversa: que transforme números binarios (originados desde el dispositivo) a caracteres (y así conseguir que los humanos los puedan comprender convenientemente). El mecanismo de "traducción" que utiliza Arduino es el conocido como "tabla ASCII", el cual consiste en una simple lista de equivalencias que vincula cada carácter de la lista con un determinado número.

La tabla ASCII tan solo lista 128 caracteres (con sus correspondientes equivalencias numéricas binarias, empezando por el 0). Esto es así porque utiliza 7 bits para codificar los caracteres, y con 7 bits solo se pueden utilizar este número de combinaciones ($2^7=128$). Los 32 primeros caracteres de la tabla ASCII se denominan "códigos de control" y son no imprimibles. De todos ellos, los más importantes –con diferencia– son el nº 0 (carácter nulo), el nº 8 (carácter retroceso), el nº 9 (tabulador), el nº 10 (salto de línea), el nº 13 (retorno de carro), el nº 27 (carácter "escape") y el nº 32 (espacio). El resto de códigos de control y todos los caracteres ASCII imprimibles (es decir, a partir del nº 33) se pueden consultar en la página <http://arduino.cc/en/Reference/ASCIIchart> (o bien en el apéndice B de este libro, el cual muestra estos códigos como referencia para el lector). Allí podemos observar, por ejemplo, que el carácter 'A' es el nº 65, el carácter 'a' el nº 97, el carácter '8' el nº 56, el carácter '#' el nº 35, etc.

Breve nota sobre ASCII, ISO-8859-1 y UTF-8

Es fácil ver que albergando tan solo 128 caracteres, la tabla ASCII no puede contener todos los símbolos existentes en los distintos lenguajes del mundo. De hecho, tan solo contiene los símbolos presentes en el idioma inglés (no aparecen ni siquiera los caracteres acentuados o la 'ñ', por ejemplo).

Un primer intento de solucionar este problema dentro del mundo de la informática fue ampliar la lista de caracteres disponibles utilizando 8 bits (en vez de los 7 bits originales) para codificar hasta $2^8=256$ combinaciones diferentes (manteniendo las primeras 128 iguales a la tabla ASCII original). A esta tabla ASCII "extendida" se le llamó codificación ISO-8859 (http://en.wikipedia.org/wiki/ISO/IEC_8859). Esta codificación en realidad tenía diferentes variantes según la lista de caracteres elegida para completar las posiciones extra: si se rellenaban, por ejemplo, con caracteres del alfabeto latino (como son, entre otros, las letras acentuadas) se estaba usando la variante ISO-8859-1 (http://en.wikipedia.org/wiki/ISO/IEC_8859-1 también conocida como ISO Latin-1), pero si se rellenaban con caracteres del alfabeto griego se estaba usando la variante ISO-8859-7, y así con otros alfabetos.

No obstante, esta solución era aún claramente insuficiente para muchos lenguajes (sobre todo no europeos). Por esta razón apareció la codificación UTF-8 (<http://en.wikipedia.org/wiki=UTF-8>, basada en el estándar de símbolos Unicode, <http://www.unicode.org>), la cual es compatible con ISO-8859-1 en los primeros 256 caracteres (y por tanto, con la tabla ASCII en los primeros 128 caracteres) pero que es capaz de contener cualquier símbolo del mundo gracias a que permite codificarlos mediante 1, 2, 3 o hasta 4 bytes (y no solamente mediante solo uno como hasta entonces). Actualmente, la codificación UTF-8 es la más usada en los programas informáticos (incluyendo el IDE Arduino) pero la placa Arduino (y muchos otros dispositivos electrónicos) tan solo son capaces de trabajar correctamente con caracteres ASCII, así que en este libro nos ceñiremos en exclusiva al uso de estos.

NOTA: Si el lector tuviera curiosidad por conocer de forma sencilla todos los símbolos Unicode (y sus respectivos números identificadores únicos asociados –llamados "code points"–) puede consultar la página <http://unicode-table.com/es>. Si quisiera saber además qué codificación UTF-8 corresponde a un determinado "code point", puede consultar <http://www.utf8-chartable.de>

Para asignar un carácter ASCII como valor de una variable de tipo *char*, deberemos tener la precaución de escribir ese carácter entre comillas simples. Por tanto, si en el sketch del ejemplo 4.1 la variable *mivariable* hubiera sido de tipo *char* en vez de *int*, para asignarle el valor de la letra A tendríamos que haber escrito una línea similar a `char mivariable='A';`.

El hecho de que los caracteres sean realmente números permite que se puedan realizar operaciones aritméticas con ellos (o mejor dicho, con su valor numérico correspondiente dentro de la tabla ASCII). Por ejemplo: si realizáramos la operación 'A' + 1 obtendríamos el valor 66, ya que en la tabla ASCII el valor numérico del carácter 'A' es 65. De hecho, inicializar una variable *char* asignándole un valor numérico en vez del carácter correspondiente es completamente equivalente (es decir: escribir `char mivariable='A';` es igual a escribir `char mivariable=65;`). Hay que tener en cuenta, no obstante, que una variable *char* puede admitir valores numéricos negativos (del -128 al -1), los cuales no tienen correspondencia con ningún carácter en la tabla ASCII.

El tipo *byte* (*unsigned char*): el valor que puede tener una variable de este tipo es siempre un número entero entre 0 y 255. Al igual que las variables de tipo *char*, las de tipo *byte* utilizan un byte (8 bits) para almacenar su valor y, por tanto, tienen el mismo número de combinaciones numéricas posibles diferentes (256), pero a diferencia de aquellas, los valores de una variable *byte* no pueden ser negativos.

El tipo *int*: el valor que puede tener una variable de este tipo es un número entero entre -32768 (-2^{15}) y 32767 ($2^{15}-1$) –si usamos una placa AVR– o un número entero entre -2147483648 (-2^{31}) y 2147483647 ($2^{31}-1$) –si usamos una placa ARM– gracias a que utilizan 2 bytes (16 bits) o 4 bytes (32 bits) de memoria para almacenarse, respectivamente (y se reserva el primer bit por la derecha para indicar el signo).

El tipo *word* (*unsigned int*): las variables de tipo *word* admiten el mismo número de combinaciones numéricas diferentes para sus posibles valores que las variables *int* porque ambos tipos ocupan el mismo espacio de memoria (2 bytes en placas AVR y 4 bytes en placas ARM), pero hay una diferencia: los valores de una variable *word* no pueden ser negativos. Es fácil ver, por tanto, que el valor que puede tener una variable *word* en las placas AVR es un número entero entre 0 y 65535 ($2^{16}-1$) y en las placas ARM un número entero entre 0 y 4294967295 ($2^{32}-1$).

El tipo *short*: el valor que puede tener una variable de este tipo para todos los modelos de placa (ya sean basadas en microcontroladores de tipo AVR o de tipo ARM) es un número entero entre -32768 (-2^{15}) y 32767 ($2^{15}-1$), gracias a que siempre utilizan 2 bytes (16 bits) de memoria para almacenarse. En este sentido, los tipos *short* e *int* para placas de la familia AVR son equivalentes, pero para las placas de tipo ARM el tipo *short* es el único que utiliza 16 bits.

El tipo *unsigned short*: las variables de tipo *unsigned short* admiten el mismo número de combinaciones numéricas diferentes para sus posibles valores que las variables de tipo *short* porque ambos tipos ocupan 2 bytes de memoria (en cualquier arquitectura, ya sea AVR o ARM), pero hay una diferencia: los valores de una variable *unsigned short* no pueden ser negativos. Es fácil ver, por tanto, que el valor que puede tener una variable de este tipo es un número entero entre 0 y 65535 ($2^{16}-1$).

El tipo *long*: el valor que puede tener una variable de este tipo para todos los modelos de placa (ya sean basadas en microcontroladores de tipo AVR o de tipo ARM) es un número entero entre -2147483648 y 2147483647 gracias a que utilizan 4 bytes (32 bits) de memoria para almacenarse. En este sentido, los tipos *long* e *int* para placas de la familia ARM son equivalentes.

El tipo *unsigned long*: el valor que puede tener una variable de este tipo para todos los modelos de placa (ya sean basadas en microcontroladores de tipo AVR o ARM) es un número entero entre 0 y 4294967295 ($2^{32}-1$). Al igual que las variables de tipo *long*, las de tipo *unsigned long* utilizan 4 bytes (32 bits) para almacenar su valor, y por tanto, tienen el mismo número de combinaciones numéricas posibles diferentes (2^{32}), pero a diferencia de aquellas, los valores de una variable *unsigned long* no pueden ser negativos (tal como ya indica su propio nombre). En este sentido, los tipos *unsigned long* y *word* para placas de la familia ARM son equivalentes.

Breve nota sobre el uso de los sistemas binario y hexadecimal

Es posible asignar a una variable entera positiva (es decir, de tipo *byte*, *word* o *unsigned long*) un valor numérico en formato binario o bien hexadecimal, además del conocido formato decimal. Los formatos binario y hexadecimal son maneras diferentes de escribir un número.

En el formato binario los números se representan utilizando solamente combinaciones de los símbolos 0 y 1. Así, por ejemplo, el número 37 (en decimal) se escribe en formato binario así: 100101.

En el formato hexadecimal, los números se representan mediante 16 símbolos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F), cada uno de los cuales es equivalente a una combinación específica de cuatro valores binarios. En concreto: 0=0000, 1=0001, 2=0010, 3=0011, 4=0100, 5=0101, 6=0110, 7=0111, 8=1000, 9=1001, A=1010, B=1011, C=1100, D=1101, E=1110 y F=1111.

El sistema binario es muy utilizado en la informática y electrónica debido al comportamiento digital que es inherente a los dispositivos tecnológicos, y el sistema hexadecimal no deja de ser una manera de "abreviar" la escritura de números binarios, ya que se utiliza para compactar su representación: por cada cuatro cifras binarias se escribe una cifra hexadecimal.

Las reglas matemáticas para pasar un número escrito en decimal a formato binario, o viceversa, o de decimal a formato hexadecimal, o viceversa, se salen fuera del alcance de este libro. Si se desean conocer, se puede consultar el artículo de la Wikipedia titulado "Sistema binario". De todas formas, estas conversiones de formato se pueden realizar rápidamente mediante la opción adecuada de la aplicación calculadora incluida por defecto en cualquier sistema operativo moderno.

Si se desea asignar a una variable un valor en formato binario, hay que preceder dicho valor mediante la letra "B" (por ejemplo, así: B10010). Si se desea asignar un valor en formato hexadecimal, hay que preceder dicho valor mediante el prefijo "0x" (por ejemplo, así: 0x54FA). Por lo tanto, una asignación posible podría ser por ejemplo: unavariable = B1010011;, donde "1010011" representa el número 83 en decimal (y el 53 en hexadecimal).

Breve nota sobre la importancia de los rangos de valores válidos

En los párrafos anteriores se ha comentado, en relación con las variables numéricas, que todas ellas tienen un rango de valores válido. Esto significa que asignar un valor fuera de este puede tener consecuencias inesperadas. Por tanto, hay que saber elegir correctamente el tipo de variable numérica que necesitaremos en cada momento. En el caso de las variables enteras, lo que uno podría pensar es utilizar siempre las de tipo *unsigned long*, las cuales admiten el rango más elevado, pero esto no es buena idea, porque cada variable de este tipo ocupa cuatro veces más que una variable *byte*, y si (por ejemplo) sabemos de antemano que los valores a almacenar no serán mayores de 100, sería una completa pérdida de memoria el utilizar variables *unsigned long*. Y la memoria del microcontrolador es uno de los recursos máspreciados y escasos para irlo desaprovechando por una mala elección de tipos.

En el supuesto caso de que un valor supere el rango válido (tanto por "encima" como por "debajo"), lo que ocurrirá es que "dará la vuelta", y el valor continuará por el otro extremo. Es decir, si tenemos una variable de tipo *byte* (por ejemplo) cuyo valor es actualmente 255 y se le suma 1, su nuevo valor será entonces 0 (si se le sumara 2, valdría entonces 1, y así). Lo mismo pasa si se supera el límite inferior:

si tenemos una variable de tipo *byte* (por ejemplo) cuyo valor actual es 0 y se le resta 1, su nuevo valor será entonces 255 (si se le restara 2, valdría entonces 254, y así). De hecho, esto lo podemos observar si ejecutamos el sketch de ejemplo 4.1 y esperamos lo suficiente: cuando los valores de la variable *mivariable* (de tipo *int*, recordemos) lleguen a su límite superior (32767), veremos cómo de forma automática el siguiente valor vuelve a ser el del principio de su rango (concretamente el -32768) para seguir aumentando sin parar otra vez hasta llegar al máximo y volver a caer de nuevo al mínimo, y así. Este fenómeno se llama "overflow".

El tipo *String*: una variable de tipo *char* solo puede almacenar un carácter individual pero no es capaz de almacenar una cadena –"string" en inglés– de caracteres (es decir, una palabra o una frase). Para ello, debemos utilizar otro tipo de variables, las de tipo *String* (notar la "S" mayúscula). Estrictamente hablando, *String* no es un "tipo de datos" sino una "clase de objeto", pero estos detalles técnicos no son relevantes ahora mismo: lo importante es saber que podemos declarar e inicializar una variable de este tipo simplemente así: `String unacadena="hola qué tal";`. También se podría declarar como *String* un carácter individual, así: `String uncaracter='a';` O inicializar una nueva variable *String* con el valor de otra ya existente: `String variablestringnueva = variablestringyaexistente;`.

Si lo que queremos es convertir valores que inicialmente eran numéricos enteros (*byte*, *int*, *word*, *long*, *unsigned long*) en variables *String* (concretamente, en su representación ASCII), deberemos utilizar la instrucción especial *String()*. Así, para declarar una variable de tipo *String* cuyo valor era inicialmente un número se debe escribir: `String unacadena=String(13);`: esto hará que tengamos una variable *String* llamada *unacadena* con el valor "13". La instrucción *String()* es muy flexible, y permite que se escriban en su interior no solamente números exactos (como el 13 del ejemplo anterior) sino también valores extraídos de variables de tipo entero o de instrucciones que devuelvan ese tipo de valores (por ejemplo: `String unacadena=String(unavariablenentera);`).

En estos casos donde hay una conversión de valores numéricos a valores *String*, se puede escribir opcionalmente dentro de la instrucción *String()* un segundo parámetro que puede valer las constantes predefinidas *BIN* o *HEX*, indicando si el valor de tipo *String* estará en formato binario o hexadecimal, respectivamente. Es decir, si por ejemplo tenemos la siguiente declaración: `String unacadena=String(45, BIN);`, el valor de *unacadena* será "00101101", y si tenemos `String unacadena=String(45, HEX);`, será "2D".

Otra característica más que podemos hacer es concatenar el valor de una variable *String* con otros valores (escritos explícitamente, o guardados en alguna variable, o devueltos por alguna instrucción) que pueden ser de tipo *String* pero también arrays de caracteres –los estudiaremos enseguida–, caracteres individuales, o incluso valores numéricos representados por sus cifras ASCII. Para ello, debemos utilizar el signo "+". Es decir, si tenemos por ejemplo las declaraciones `String unacadena="hola"; char uncaracter='o'; y byte unnumero=123;` para asignar a *unacadena* el valor "holao123" podemos hacer simplemente `unacadena= unacadena + uncaracter + unnumero;` (sobrescribiendo el valor que tenía anteriormente, lógicamente). También se lo podríamos asignar a una tercera variable *String* diferente (llamémosla *otramas*): `otramas = unacadena + uncaracter + unnumero;.` La concatenación de cadenas es muy útil cuando se quiere mostrar una combinación de valores junto con su descripción en una sola cadena (en una pantalla LCD, en una conexión Ethernet o serie, etc.).

Podemos hacer incluso dos cosas a la vez: declarar una variable *String* e inicializarla en ese mismo momento con la unión de varias cadenas independientes. En este caso, además del operador "+", necesitaremos utilizar también la instrucción *String()*. Un ejemplo: para crear la variable *unacadena* con el contenido "una frase, otra frase" deberíamos escribir: `String unacadena=String("una frase," + " otra frase");`

El tipo *float*: el valor que puede tener una variable de este tipo es un número decimal. Ocupa 4 bytes de memoria. Los valores *float* posibles pueden ir desde el número $-3,4028235 \cdot 10^{38}$ hasta el número $3,4028235 \cdot 10^{38}$. Debido a este grandísimo rango de valores, los números decimales son usados frecuentemente para aproximar valores analógicos continuos. No obstante, solo tienen 6 o 7 dígitos de precisión en total (es decir, contando además de los de la parte decimal, los de la parte entera). Así pues, los valores *float* no son exactos, y pueden producir resultados inesperados, como por ejemplo que $6.0/3.0$ no dé exactamente 2.0.

Otro inconveniente de los valores de tipo *float* es que el cálculo matemático con ellos es mucho más lento que con valores enteros, por lo que debería evitarse el uso de valores *float* en partes de nuestro sketch que necesiten ejecutarse a gran velocidad.

Los números decimales se han de escribir en nuestro sketch utilizando la notación anglosajona (es decir, utilizando el punto decimal en vez de la coma). Si lo deseamos, también se puede utilizar la notación científica (es decir, el número 0,0234 –equivalente a $2,34 \cdot 10^{-2}$ – lo podríamos escribir como $2.34e-2$).

EL MUNDO GENUINO-ARDUINO

El tipo **double**: para las placas de tipo AVR es un sinónimo exactamente equivalente del tipo *float*, y por tanto, no aporta ningún aumento de precisión respecto a este. Para las placas de tipo ARM, en cambio, aporta el doble de precisión ya que utiliza 8 bytes (64 bits) para almacenar su valor en memoria.

A modo de resumen, para que el lector se haga una idea más concisa de las características ofrecidas por los distintos tipos de datos comentados en los párrafos anteriores, a continuación ponemos a su disposición una tabla de rápida consulta mostrando la cantidad de memoria ocupada y el rango de valores admitido para cada tipo de dato simple.

Tipo de dato	Cantidad de memoria ocupada	Rango de valores
<i>boolean</i>	1 byte	0 – <i>false</i> – o 1 – <i>true</i> –
<i>char</i>	1 byte	-128 a 127
<i>byte (unsigned char)</i>	1 byte	0 a 255
<i>short</i>	2 bytes	-32768 a 32767
<i>unsigned short</i>	2 bytes	0 a 65535
<i>int</i>	2 bytes (AVR) 4 bytes (ARM)	-32768 a 32767 -2147483648 a 2147483647
<i>word (unsigned int)</i>	2 bytes (AVR) 4 bytes (ARM)	0 a 65535 0 y 4294967295
<i>long</i>	4 bytes	-2147483648 a 2147483647
<i>unsigned long</i>	4 bytes	0 y 4294967295
<i>float</i>	4 bytes	$-3,4028235 \cdot 10^{38}$ a $3,4028235 \cdot 10^{38}$
<i>double</i>	4 bytes (AVR) 8 bytes (ARM)	$-3,4028235 \cdot 10^{38}$ a $3,4028235 \cdot 10^{38}$ $-1,7 \cdot 10^{308}$ a $1,7 \cdot 10^{308}$

Tipos complejos

El tipo **array**: este tipo de datos en realidad no existe como tal, sino que lo que existen son arrays de variables de tipo *boolean*, arrays de variables de tipo *int*, arrays de variables de tipo *float*, etc. En definitiva: arrays de variables de cualquier tipo de los conocidos. Un array (también llamado "vector") es una colección de variables de un tipo concreto que tienen todas el mismo y único nombre, pero que pueden distinguirse entre sí por un número a modo de índice. Es decir: en vez de tener diferentes variables –por ejemplo de tipo *char*– cada una independiente de las demás (*varChar1*, *varChar2*, *varChar3*...) podemos tener un único array que las agrupe todas bajo un mismo nombre (por ejemplo, *varChar*), y que permita que cada variable pueda manipularse por separado gracias a que dentro del array cada una está identificada mediante un índice numérico, escrito entre corchetes

(*varChar[0]*, *varChar[1]*, *varChar[2]*...). Los arrays sirven para ganar claridad y simplicidad en el código, además de facilitar la programación.

Podemos crear –declarar– un array (ya sea en la zona de declaraciones globales o bien dentro de alguna sección concreta), de las siguientes maneras:

```
int varInt[6];
```

Declara un array de 6 elementos (es decir, variables individuales) sin inicializar ninguno.

```
int varInt[] = {2,5,6,7};
```

Declara un array sin especificar el número de elementos. No obstante, se asignan (entre llaves, separados por comas) los valores directamente a los elementos individuales, por lo que el compilador es capaz de deducir el número de elementos total del array (en el ejemplo de la derecha, cuatro).

```
int varInt[8] = {2,5,6,7};
```

Declara un array de 8 elementos e inicializa algunos de ellos (los cuatro primeros), dejando el resto sin inicializar. Lógicamente, si se inicializaran más elementos que lo que permite el tamaño del array (por ejemplo, si se asignan 9 valores a un array de 8 elementos), se produciría un error.

```
char varChar[6] = "hola";
char varChar[6] = {'h','o','l','a'};
char varChar[] ="hola";
```

La primera forma declara e inicializa un array de seis elementos de tipo *char*. Tal como se puede ver en ella, esta clase de arrays tiene la

particularidad de poderse inicializar indicando como valor una cadena de caracteres en forma de palabra o frase escrita entre comillas dobles. Pero también se pueden declarar como un array "estándar" de elementos individuales, que es como muestra la segunda forma. Observar en este caso la diferencia de comillas: un carácter siempre se especifica entre comillas simples pero una cadena siempre se especifica entre comillas dobles. También es posible, tal como muestra la tercera forma, declarar una cadena sin necesidad de especificar su tamaño (ya que el compilador lo puede deducir a partir del número de elementos –es decir, de caracteres– inicializados).

Hay que tener en cuenta que el primer valor del array tiene el índice 0 y por tanto, el último valor tendrá un índice igual al número de elementos del array menos uno. Cuidado con esto, porque asignar valores más allá del número de elementos

EL MUNDO GENUINO-ARDUINO

declarados del array es un error. En concreto, si por ejemplo tenemos un array de 2 elementos de tipo entero (es decir, declarado así: int varInt[2];), para asignar un nuevo valor (por ejemplo, 27) a su primer elemento deberíamos escribir: varInt[0] = 27; , y para asignar el segundo valor (por ejemplo, 17), escribiríamos varInt[1] = 17;. Pero si asignáramos además un tercer valor así, varInt[2] = 53; , cometeríamos un error porque estaríamos sobre pasando el final previsto del array (y por tanto, utilizando una zona de la memoria no reservada, con resultados imprevisibles).

En el caso concreto de los arrays de caracteres hay que tener en cuenta, además, una particularidad muy importante: este tipo de arrays siempre deben ser declarados con un número de elementos una unidad mayor que el número máximo de caracteres que preveamos guardar. Es decir, si se va a almacenar la palabra "hola" (de cuatro letras), el array deberá ser declarado como mínimo de 5 elementos. Esto es así porque este último elemento siempre se utiliza para almacenar automáticamente un carácter especial (el carácter "nulo", con código ASCII 0), que sirve para marcar el final de la cadena. Esta marca es necesaria para que el compilador sepa que la cadena ya terminó y no intente seguir leyendo más posiciones. Si no sabemos de antemano cuál es la longitud del texto que se guardará en un array de caracteres, podemos declarar este con un número de elementos lo suficientemente grande como para que haya elementos sin ser asignados, aun sabiendo que así posiblemente estaremos desaprovechando memoria del microcontrolador.

Por otro lado, no sobra comentar que, además de asignar un valor explícito a un elemento de un array (tal como se acaba de comentar en los párrafos anteriores), también es posible asignarle el valor que tenga en ese momento otra variable independiente (preferiblemente del mismo tipo). Por ejemplo, mediante la línea varInt[4]=x; estaremos asignando el valor actual de una variable llamada x al quinto elemento (el índice empieza por 0!) del array llamado *varInt*. Y a la inversa también se puede: para asignar el valor que tenga en ese momento el quinto elemento del array *varInt* a la variable independiente x, simplemente deberemos ejecutar la línea: x=varInt[4]; .

Breve nota sobre los arrays de caracteres y el tipo de datos *String*

El lector seguramente se estará preguntando por qué el lenguaje Arduino ofrece dos maneras diferentes de inicializar variables cuyo contenido (cadenas de caracteres) es el mismo: o bien declarándolas como *String*, o bien como un array de tipo *char*. La razón radica en la diferencia con la que el microcontrolador de la placa Arduino gestiona internamente, según si una variable es de un tipo u otro, el uso de la memoria SRAM que esta requiere. Más en concreto (y simplificando

mucho), la ubicación de los valores de tipo *String* dentro de dicha memoria puede ser cambiada por el microcontrolador automáticamente durante la ejecución del sketch (esto es lo que se llama "asignación dinámica") pero la ubicación de los arrays de tipo *char*, en cambio, no. Esto hace que un valor de tipo *String* pueda estar guardado en un momento dado en un lugar de la memoria SRAM y en otro momento en otro lugar, dependiendo sobre todo del tamaño –cambiante– que pueda tener ese valor a lo largo de la ejecución del sketch. Esto implica que el microcontrolador debe realizar un trabajo suplementario con los valores *String* (para no corromperlos en las sucesivas (re)asignaciones de lugares de memoria) que no existe en el manejo de los arrays de tipo *char*. A cambio, utilizar variables de tipo *String* en nuestros sketches nos permite manipular las cadenas de caracteres de una forma mucho más sencilla y cómoda (tal como pronto veremos). Dicho esto, se deja al lector que compruebe por sí mismo la conveniencia de utilizar un tipo de cadenas u otro.

En el caso de escribir una cadena en nuestro sketch directamente (por ejemplo, como valor de algún parámetro de alguna instrucción, tal como `Serial.print("Una cadena");`), esta cadena siempre será tratada por el microcontrolador como array de caracteres.

En el caso de trabajar con grandes cantidades de texto (por ejemplo, en un proyecto con pantallas LCD), a menudo es conveniente utilizar arrays de cadenas. Para declarar un array de esta clase, se puede optar simplemente por escribir una línea tal como `String varCadenas[]={"Cad0","Cad1","Cad2","Cad3"};`, donde se estaría creando un array de variables de tipo *String*. Pero también es posible crear un array de arrays de caracteres; en este caso, se ha de emplear el tipo de datos especial *char** (notar el asterisco final), así: `char* varCadenas[]{"Cad0","Cad1","Cad2","Cad3"};`.

Breve nota sobre los punteros

En realidad, el asterisco en la declaración anterior indica que en realidad estamos declarando un array de "punteros". Los punteros son unos elementos del lenguaje Arduino (provenientes del lenguaje C en el que está basado) muy potentes pero a la vez ciertamente complejos. En este libro no se tratarán, debido a que sus posibles usos son avanzados y pueden confundir al lector que se inicia en la programación: lo único que necesitaremos saber es cómo se declaran los arrays de arrays de caracteres y nada más. La buena noticia es que, una vez declarado el array de cadenas, podemos trabajar con él (asignando valores a sus elementos, consultándolos, etc.) como cualquier otro tipo de array sin notar para nada que estamos usando punteros.

EL MUNDO GENUINO-ARDUINO

También es posible definir arrays "bidimensionales" (es decir, arrays cuyos elementos son a su vez arrays de un tipo de datos dado). Este tipo de arrays se pueden entender como una "tabla de valores". Por ejemplo, una línea como `int arrayBi[3][2];` está declarando un array de 3 elementos donde cada uno de ellos es también un array (lo llamaremos "subarray"), pero de 2 elementos de tipo entero. Si quisieramos inicializar este array bidimensional, la sintaxis a seguir consiste en agrupar entre llaves (`{ } y { }`) los elementos de cada "subarray", separándolos entre sí por comas, y agrupar finalmente todo el conjunto dentro de unas llaves globales. Por ejemplo, así: `int arrayBi[3][2] = {{2,7},{4,9},{8,1}};`. Si realizamos la inicialización anterior, el elemento `arrayBi[0][1]` contendrá entonces el valor 7 –porque del primer "subarray" es segundo elemento (recordemos que las posiciones empiezan por 0)– o, por poner otro ejemplo, el elemento `arrayBi[2][0]` contendrá el valor 8 –porque del tercer "subarray" es el primer elemento–.

El tipo struct: en alguna ocasión nos puede interesar agrupar bajo un nombre común varios valores que no tienen por qué ser del mismo tipo (pero que sí tienen alguna relación de significado entre ellos). En estos casos un array no nos sirve, así que debemos emplear otro elemento del lenguaje Arduino llamado *struct* (o "estructura"). Usar estructuras (es decir, "agrupaciones de variables de –posiblemente– diverso tipo") en vez de variables independientes mejora la organización de nuestro código, lo que redunda en un mayor orden y claridad de nuestros sketches, y por tanto, en un desarrollo más rápido y eficaz.

Una estructura es, en realidad, un tipo de dato personalizado. Es decir, antes de poder usar cualquier variable de tipo estructura en nuestro código debemos especificar cómo será dicha estructura concreta (es decir, por qué conjunto de variables individuales estará formada). Una vez hecho esto, ya será posible declarar (y/o inicializar) tantas variables de ese tipo (es decir, con esa estructura) como sea necesario (las cuales llamaremos a partir de ahora "variables-contenedor"). A partir de entonces, cada una de estas variable-contenedor podrá almacenar un conjunto de valores, cada uno de los cuales se corresponderá con cada una de las variables individuales que forman su estructura.

Por ejemplo, si disponemos de un sensor que puede medir tanto la temperatura como la humedad ambiental (existen en el mercado varios sensores de este tipo), en vez de utilizar en nuestro sketch una variable independiente llamada (por ejemplo) *temperatura* para guardar el valor obtenido de dicha magnitud y otra variable diferente llamada *humedad* para hacer lo propio con el valor correspondiente, podríamos definir una estructura llamada (por ejemplo) *sensor* que incluyera dentro de sí la declaración de esas variables individuales *temperatura* y

humedad. Una vez definida la estructura *sensor*, a continuación podríamos declarar (indicando como su tipo de datos precisamente dicha estructura) las variables-contenedor necesarias (llamémoslas *s1*, *s2*, etc.) o mejor, un array de variables-contenedor (llamémoslo *s[]*) y proceder entonces al relleno de valores concretos (ya sea en la inicialización o posteriormente).

Para crear una nueva estructura (es decir, un nuevo tipo de datos) se ha de escribir la línea *struct nombreEstruct {...};*; donde los puntos suspensivos indican que allí ha de haber las declaraciones de las diferentes variables individuales que la forman. A partir de aquí, la declaración de una variable-contenedor que albergue ese conjunto de variables individuales se puede realizar simplemente indicando como su tipo de datos el nombre de la estructura recién creada, así: *nombreEstruct nombreVarCont;* (también podríamos haber declarado un array de –por ejemplo, 4– variables-contenedor simplemente escribiendo *nombreEstruct nombreVarCont[4];*). Si en el momento de declarar una variable-contenedor quisiéramos, además, inicializar las variables individuales contenidas en ella, deberíamos indicar sus valores iniciales entre llaves, así: *nombreEstruct nombreVarCont = {valorVar1, valorVar2,...};* donde el orden de los valores debe ser el mismo que el orden de las declaraciones de las variables individuales dentro de la estructura. Si, por el contrario, quisiéramos (re)asignar los valores de todas esas variables individuales después de haber declarado la variable-contenedor, entonces deberíamos escribir: *nombreVarCont = {valorVar1, valorVar2, ...}.* También podemos modificar el valor de solo una determinada variable individual presente dentro de una determinada variable-contenedor escribiendo *nombreVarCont.nombreVarInd = valorVarInd* . En cualquier caso, para acceder al valor que tiene en un momento preciso una determinada variable individual que forma parte de una determinada variable-contenedor, la notación a utilizar es *nombreVarCont.nombreVarInd*.

El siguiente código es una variación respecto al ejemplo 4.1 donde, en vez de mostrar por el "Serial monitor" el valor cambiante de una variable de tipo *int*, se muestran los valores de algunas de las variables individuales contenidas dentro de dos variables-contenedor (*s1* y *s2*), ambas de tipo *sensor*, tipo que es definido como una estructura de tres datos (cadena, decimal y entero):

Ejemplo 4.2

```
struct sensor { //Defino el tipo de datos 'sensor', estructura formada por tres variables individuales
    char* nombreSensor;
    /*Los variables de dentro de una estructura que vayan a almacenar cadenas de caracteres no se pueden
    declarar de la manera habitual (usando corchetes) sino de esta otra manera (con un asterisco) */
    float humedad;
```

```

int temp;
}; //Notar el punto y coma!
sensor s[2]; //Declaro un array conteniendo dos variables-contenedor de tipo 'sensor' (sin inicializar)
void setup() {
    Serial.begin(9600);
    //Asigno un valor inicial solo a la variable 'nombreSensor' del primer elemento del array 's'
    s[0].nombreSensor="Sensor1";
    //Inicializo completamente el segundo elemento del array 's'
    s[1] = {"Sensor2",0.71,28};
}
void loop() {
    Serial.println(s[0].nombreSensor); //Muestra 'Sensor1'
    Serial.println(s[0].humedad); //Muestra el valor por defecto sin inicializar (0.00 en este caso)
    Serial.println(s[1].temp); //Muestra 28,29,30,31,32,33,34....(ver línea posterior)
    s[1].temp = s[1].temp + 1; //Modifico el valor actual de 'temp' aumentándolo una unidad
}

```

Existen otros tipos de datos complejos como (entre otros) las uniones o las enumeraciones pero en este libro no se estudiarán por ser demasiado avanzados y/o específicos. De todas formas, si el lector lo desea, puede encontrar una breve explicación de las enumeraciones en <http://playground.arduino.cc/Code/Enum>.

La instrucción *sizeof()*

Elegir adecuadamente el tipo de cada variable usada en nuestros sketches (*int*, *float*, *char...*) es importante para que estos no adquieran un tamaño excesivo (y en casos extremos, no quepan en la memoria del microcontrolador). Para conocer en un momento determinado el espacio ocupado en memoria Flash por una variable de cualquier tipo podemos usar la instrucción *sizeof()*. Esta instrucción tiene como único parámetro el nombre de la variable de la cual queremos saber el tamaño que ocupa. Lógicamente, si esa variable fuera, por ejemplo, de tipo *char*, *sizeof()* devolverá 1, si fuera de tipo *int* devolverá 2, y así sucesivamente. No parece, por tanto, que *sizeof()* sea en general demasiado útil, pero hay un caso en que sí: cuando queremos controlar el tamaño de los arrays. Expliquemos esto.

Tal como se puede ver en el siguiente código de ejemplo, si declaramos un array definiéndole un número de elementos predeterminado, el tamaño del array siempre será el resultado de la operación *nºelementospdefinidos*tamañoelemento*, aunque sus elementos no estén inicializados (es decir, aunque no tengan ningún valor concreto). En cambio, si declaramos un array sin número de elementos predeterminado, el tamaño del array será el resultado de la operación *nºelementosinicializadosenesemomento*tamañoelemento*:

Ejemplo 4.3

```

void setup() {
    int a[8] = {2,5,6,7};
    int b[6];
    int c[] = {2,5,6,7};
    char d[6] = "holá";
    char e[6] = {'h','o','l','a'};
    char f[] = "holá";
    Serial.begin(9600);
    Serial.println(sizeof(a)); //Muestra 16 = 8 x 2bytes
    Serial.println(sizeof(b)); //Muestra 12 = 6 x 2bytes
    Serial.println(sizeof(c)); //Muestra 8 = 4 x 2bytes
    Serial.println(sizeof(d)); //Muestra 6 = 6 x 1byte
    Serial.println(sizeof(e)); //Muestra 6 = 6 x 1byte
    Serial.println(sizeof(f)); //Muestra 5 = 4 x 1byte + 1byte correspondiente al carácter 'nulo' final
}
void loop() {
    //No se ejecuta nada aquí
}

```

Una utilidad "colateral" del uso de `sizeof()` con arrays (siempre que estos no hayan sido declarados con un número fijo de elementos) es que nos permite saber cuántos elementos están inicializados en un determinado momento. Para ello, debemos emplear la expresión `sizeof(array)/sizeof(array[0])`. Tal como se puede ver, en realidad la expresión anterior simplemente divide el tamaño total del array entre el tamaño de un solo elemento, obteniendo por tanto el número de elementos del array. Esta expresión nos será útil en los casos donde queramos recorrer (mediante un bucle) todos los elementos de un array, desde el primero hasta el último definido, uno tras otro.

La instrucción `sizeof()` también puede servir para conocer el tamaño real de una variable-contenedor de tipo estructura, ya que aunque uno podría pensar que ese tamaño podría saberse sumando el tamaño de las variables individuales que forman dicha estructura, debido a la gestión y optimización interna de memoria que realizan los microcontroladores de las placas Arduino, esto no siempre es así.

Cambio de tipo de datos (numéricos)

No se puede cambiar nunca el tipo de una variable: si esta se declaró de un tipo concreto, seguirá siendo de ese tipo a lo largo de todo el sketch. Pero lo que sí se puede hacer es cambiar "al vuelo" en un momento concreto el tipo del valor que contiene. Esto se llama "casting", y puede ser útil cuando se quiere utilizar ese valor

EL MUNDO GENUINO-ARDUINO

en cálculos que requieran un tipo diferente del original, o también cuando se quiere asignar el valor de una variable de un tipo a otra variable de tipo diferente. Para convertir un valor –del tipo que sea– en otro, podemos utilizar alguna de las siguientes instrucciones (tal como ilustra el código de ejemplo posterior):

`char()`: escribiremos entre () el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo *char*.

`byte()`: escribiremos entre () el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo *byte*.

`int()`: escribiremos entre () el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo *int*.

`word()`: escribiremos entre () el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo *word*.

`long()`: escribiremos entre () el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo *long*.

`float()`: escribiremos entre () el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo *float*.

Ejemplo 4.4

```
float variablefloat=3.4;
byte variablebyte=126;
void setup() {
    Serial.begin(9600);
    Serial.println(byte(variablefloat));
    Serial.println(int(variablefloat));
    Serial.println(word(variablefloat));
    Serial.println(long(variablefloat));
    Serial.println(char(variablebyte));
    Serial.println(float(variablebyte));
}
void loop() { }
```

Se puede comprobar, una vez ejecutado el sketch anterior, que todas las conversiones del valor *float* a valores de tipo entero (indiferentemente de si son *byte*, *int*, *word* o *long*) truncan el resultado: pasamos de tener un 3,4 a tener un 3. La diferencia está en los bytes que ocupa ese 3 en la memoria. También podemos comprobar que al convertir un valor numérico a tipo *char*, se muestra el carácter correspondiente de la tabla ASCII. Finalmente, la conversión de un valor entero en un valor decimal provoca que podamos trabajar precisamente con decimales a partir de entonces.

Una situación concreta donde debemos controlar que los tipos de las variables sean los correctos es en el cálculo matemático. Por ejemplo, si se ejecuta el siguiente código, se puede ver que el resultado obtenido es 2 cuando debería ser 2,5:

Ejemplo 4.5

```
float resultado;
int numerador=5;
int denominador=2;
void setup() {
    Serial.begin(9600);
    resultado=numerador/denominador;
    Serial.println(resultado);
}
void loop() {}
```

¿Por qué pasa esto? Porque tanto las variables *numerador* como *denominador* son enteras, y por tanto, el resultado siempre será entero, a pesar de que lo guardemos en una variable de tipo *float*. Esto es así porque el resultado de un cálculo matemático donde intervienen diferentes tipos enteros siempre es del tipo entero con mayor uso de memoria y con signo –para no perder información– pero nunca es decimal. Solo si interviene en el cálculo algún valor de tipo decimal, entonces el resultado será de tipo decimal (aunque seguirá siendo responsabilidad nuestra guardar ese valor en una variable de tipo decimal para que no haya truncamientos).

Para solucionar el problema del código anterior, hemos de hacer un "casting" a *float* de uno de los dos elementos de la división (o de los dos si queremos) para que la operación se realice usando esos nuevos tipos y se obtenga un resultado ya "retipeado", listo para ser asignado a la variable de tipo decimal. Es decir, podríamos sustituir `resultado=numerador/denominador;` por `resultado=float(numerador)/denominador;`

Otro problema relacionado con el cambio de tipos de datos lo podemos observar en el siguiente código:

Ejemplo 4.6

```
int numero=100;
long resultado;
void setup() {
    Serial.begin(9600);
    resultado=numero*1000;
    Serial.println(resultado);
}
void loop() {}
```

EL MUNDO GENUINO-ARDUINO

Al ejecutar el código anterior veremos por el "Serial monitor" un número que no es el resultado correcto de multiplicar 100 por 1000. ¿Por qué, si la variable *resultado* es de tipo *long* y por tanto puede almacenar un número de esta magnitud? Porque el valor que se le asigna es el resultado de multiplicar el valor de *numero* (de tipo *int*) por el número 1000 (que, si no se especifica explícitamente, es de tipo *int* también ya que hemos de saber que todo número escrito literalmente es siempre de tipo *int*). Es decir, se multiplican dos valores de tipo *int*, y el resultado por tanto sí que sobrepasa el rango aceptado por ese tipo de datos. Que la variable *resultado* sea de tipo *long* no influye para que el número obtenido de la multiplicación de dos números *int* deje de ser incorrecto, porque el cálculo de la multiplicación se realiza antes de asignar el resultado a *resultado*. Es decir, cuando se asigna el valor a *resultado*, este ya ha sufrido el "overflow". Para evitar esto, lo más fácil sería forzar a que alguno de los elementos presentes en el cálculo sea del mismo tipo que el de la variable *resultado* porque, tal como ya hemos comentado anteriormente, el tipo de datos del número resultante es el mismo que el tipo de datos del operando con mayor capacidad. Por tanto, si por ejemplo hacemos que la variable *numero* sea *long*, el resultado de la multiplicación será de tipo *long*, con lo que no se producirá "overflow" y se asignará finalmente de forma correcta a la variable *resultado*. Es decir:

Ejemplo 4.6 (BIS)

```
int numero=100;
long resultado;
void setup() {
  Serial.begin(9600);
  resultado=long(numero)*1000;
  Serial.println(resultado);
}
void loop() {}
```

Ya sabemos que cuando dentro del código Arduino escribimos directamente números, se asume por defecto que son de tipo *int*. No obstante, si tras su valor literal añadimos la letra *U* (sin comillas), su tipo por defecto será *word*, si añadimos *L*, su tipo será *long* y si se añadimos *UL*, su tipo será *unsigned long*. Es decir, una línea como *resultado=numero*1000L*; hubiera conseguido el mismo efecto que la línea existente en el código anterior.

CONSTANTES

Es posible declarar una variable de tal forma que consigamos que su valor (del tipo que sea) permanezca siempre inalterado. Es decir, que su valor no se pueda

modificar nunca porque esté marcado como de "solo lectura". De hecho, a este tipo de variables ya no se les llama así por motivos obvios, sino "constantes". Las constantes se pueden utilizar como cualquier variable de su mismo tipo, pero si se intenta cambiar su valor, el compilador lanzará un error.

Para convertir una variable (sea global o local) en constante, lo único que hay que hacer es preceder la declaración de esa variable con la palabra clave *const*. Por ejemplo, para convertir en constante una variable llamada *sensor* de tipo *byte*, simplemente se ha de declarar así: `const byte sensor;`.

Existe otra manera de declarar constantes en el lenguaje Arduino, que es utilizando la directiva especial `#define` (heredada del lenguaje C). No obstante, se recomienda el uso de *const* por su mayor flexibilidad y versatilidad.

PARÁMETROS DE UNA INSTRUCCIÓN

Antes de empezar a aprender y a utilizar las diferentes instrucciones que ofrece el lenguaje Arduino, debemos tener claro un concepto fundamental: los parámetros de una instrucción. Ya se habrá observado que, en los sketches de ejemplo anteriores, al final del nombre de cada instrucción utilizada (*Serial.begin()*, *Serial.println()*, etc) siempre aparecen unos paréntesis. Estos paréntesis pueden estar vacíos pero también pueden incluir en su interior un número/letra/palabra, o dos, o tres, etc. (si hay más de un valor han de ir separados por comas). Cada uno de estos valores es lo que se denomina un "parámetro", y el número de ellos y su tipo (que no tiene porqué ser el mismo para todos) dependerá de cada instrucción en particular.

Los parámetros sirven para modificar el comportamiento de la instrucción en algún aspecto. Es decir: las instrucciones que no tienen parámetros hacen una sola función (su tarea encomendada) y punto: no hay posibilidad de modificación porque siempre harán lo mismo de la misma manera. Cuando una instrucción, en cambio, tiene uno o más parámetros, también hará su función preestablecida, pero la manera concreta vendrá dada por el valor de cada uno de sus parámetros, los cuales modificarán alguna característica concreta de esa acción a realizar.

Por ejemplo: supongamos que tenemos una instrucción llamada *lalala()* que tiene (que "recibe", técnicamente hablando) un parámetro. Supongamos que si el parámetro vale 0 (es decir, si se escribe *lalala(0);*), lo que hará esta instrucción será imprimir por pantalla "Hola, amigo"; si el parámetro vale 1 (es decir, escribiendo *lalala(1);*), lo que hará será imprimir por pantalla "¿Qué tal estás?" y si el parámetro

EL MUNDO GENUINO-ARDUINO

vale 2 (es decir, escribiendo *lalala(2);*) imprimirá "Muy bien", etc. Evidentemente, en las tres posibilidades el comando *lalala()* hace esencialmente lo mismo: imprimir por pantalla (para eso es un comando, sino podríamos estar hablando de tres comandos diferentes), pero dependiendo del valor de su único parámetro, la acción de imprimir por pantalla un mensaje es modificada en cierta manera. Resumiendo: las instrucciones serían como los verbos (ejecutan órdenes), y los parámetros serían como los adverbios (dicen de qué manera se ejecutan esas órdenes). Observar, por otro lado, que en este ejemplo hipotético se está utilizando un parámetro numérico; no valdría en este caso dar otro tipo de valor (como una letra) porque la instrucción *lalala()* no estaría preparada para recibirla y daría error. Cada parámetro ha de tener un tipo de datos predefinido.

VALOR DE RETORNO DE UNA INSTRUCCIÓN

Otro concepto fundamental relacionado con las instrucciones del lenguaje Arduino es el de "valor de retorno". Las instrucciones, además de recibir parámetros de entrada (si lo hacen), y aparte de hacer la tarea que tienen que hacer, normalmente también devuelven un valor de salida (o "de retorno"). Un valor de salida es un dato que podemos obtener en nuestro sketch como resultado "tangible" de la ejecución de la instrucción. El significado de ese valor devuelto dependerá de cada instrucción concreta: algunos son de control (indicando si la instrucción se ha ejecutado bien o mal), otros son resultados numéricos obtenidos tras la ejecución de algún cálculo matemático, etc.

Ya sabemos que cada vez que se llega a una línea en nuestro sketch donde aparece el nombre de la instrucción con sus posibles parámetros, se ejecuta. Lo que no sabíamos es que el valor devuelto de esa ejecución automáticamente "sustituye" dentro del código al nombre de la instrucción y con esta sustitución ya hecha se continúa ejecutando el resto de la línea. Para entenderlo mejor, supongamos que tenemos una instrucción llamada *lalala()* que devuelve un determinado valor. Si queremos usar el valor devuelto podemos:

Asignar ese valor a una variable del mismo tipo, y así poder usarlo posteriormente. Es decir, si esa variable la llamáramos por ejemplo *unavariable*, deberíamos escribir algo parecido a: *unavariable=lalala();*. Fijarse que en este caso, tal como hemos dicho, una vez ejecutada la instrucción, su nombre es "sustituido" por su valor devuelto, y seguidamente este es asignado a *unavariable*.

Utilizar ese valor (que insistimos: "sustituye" al nombre de la instrucción allí donde esté escrito cuando esta se ejecuta) **directamente dentro de otra instrucción**. Por ejemplo, para ver el valor devuelto por *lalala()* podríamos ejecutarla y enviar al "Serial monitor" ese valor, todo de una sola vez, así: `Serial.println(lalala());`

Si no se desea utilizar en ningún momento el valor devuelto, no hace falta hacer nada especial: simplemente ejecutar la instrucción de la forma habitual y listo.

LA COMUNICACIÓN SERIE CON LA PLACA ARDUINO

Ya hemos explicado en capítulos anteriores que el microcontrolador ATmega328P dispone de un receptor/transmisor serie de tipo TTL-UART que permite comunicar la placa Arduino UNO con otros dispositivos (normalmente, nuestro computador), para así poder transferir datos entre ambos. El canal físico de comunicación en estos casos suele ser el cable USB, pero también pueden ser los pines digitales 0 (RX) y 1 (TX) de la placa. Si se usan estos dos pines para comunicar la placa con un dispositivo externo, tendremos que conectar concretamente el pin TX de la placa con el pin RX del dispositivo, el RX de la placa con el TX del dispositivo y compartir la tierra de la placa con la tierra del dispositivo. Hay que tener en cuenta, no obstante, que si se utilizan estos dos pines para la comunicación serie, no podrán ser usados entonces como entradas/salidas digitales estándar.

Dentro de nuestros sketches podemos hacer uso de este receptor/transmisor TTL-UART para enviar datos al microcontrolador (o recibirllos de él) gracias al elemento del lenguaje Arduino llamado *Serial*. En realidad, *Serial* es lo que llamamos un "objeto" del lenguaje. Los objetos son entidades que representan elementos concretos de nuestro sketch. El concepto de objeto es algo abstracto, pero para entenderlo mejor, simplemente supondremos que son "contenedores" que agrupan diferentes instrucciones con alguna relación entre ellas. Por ejemplo, el objeto *Serial* representa por sí mismo una comunicación serie establecida con la placa, y en nuestro sketch podremos hacer uso de un conjunto de instrucciones disponibles dentro de él que sirven para manipular dicha comunicación serie. Si utilizáramos dos objetos de clase *Serial*, podríamos manejar entonces dos conexiones serie diferentes, y cada una sería controlada mediante las instrucciones de su objeto respectivo.

Las instrucciones existentes dentro de un objeto (no todas las instrucciones del lenguaje Arduino pertenecen a un objeto) se escriben siguiendo la sintaxis *nombreObjeto.nombreInstrucción()*. Por eso las instrucciones utilizadas en el sketch

EL MUNDO GENUINO-ARDUINO

de ejemplo del principio de este capítulo, al pertenecer al objeto *Serial*, tienen nombres como *Serial.begin()* o *Serial.println()*.

A continuación, explicaremos la sintaxis, el funcionamiento y la utilidad de las instrucciones incluidas en el objeto *Serial*. Empezaremos por una ya conocida:

Serial.begin(): abre el canal serie para que pueda empezar la comunicación por él. Por tanto, su ejecución es imprescindible antes de realizar cualquier transmisión por dicho canal. Por eso normalmente se suele escribir dentro de la sección *void setup()*. Además, mediante su primer parámetro –de tipo *long* y el único obligatorio–, especifica la velocidad en bits/s a la que se producirá la transferencia serie de los datos. Para la comunicación con un computador, se suele usar el valor de 9600, pero se puede indicar cualquier otra velocidad. Lo que sí es importante es que el valor escrito como parámetro coincida con el que se especifique en el desplegable que aparece en el "Serial monitor" del IDE de Arduino, o si no la comunicación no estará bien sincronizada y se mostrarán símbolos sin sentido. Esta instrucción no tiene valor de retorno.

Serial.begin() admite también un segundo parámetro (opcional) cuyo valor ha de ser el nombre de una determinada constante predefinida, la cual sirve para indicar ciertas características técnicas que la comunicación serie a punto de poner en marcha debe cumplir (concretamente, su número de "bits de datos", su número de "bits de paridad" y su número de "bits de parada"). Si no se especifica ningún valor, *Serial.begin()* emplea por defecto la constante *SERIAL_8N1*, que equivale a 8, 0 y 1 bit, respectivamente. Para conocer las demás constantes posibles (todas de nombre *SERIAL_XXX*) emplazo al lector a la documentación online oficial de esta instrucción.

También existe la instrucción **Serial.end()**, la cual no tiene ningún argumento ni devuelve nada, y que se encarga de cerrar el canal serie; de esta manera, la comunicación serie se deshabilita y los pines RX y TX vuelven a estar disponibles para la entrada/salida general. Para reabrir el canal serie otra vez, se debería usar de nuevo *Serial.begin()*.

La otra instrucción que hemos utilizado en los sketches de ejemplo anteriores es *Serial.println()*, que pertenece a un conjunto de instrucciones que permiten enviar datos desde el microcontrolador hacia su exterior. Estudiémoslas en su conjunto.

Instrucciones para enviar datos desde la placa al exterior

Serial.print(): envía desde el microcontrolador hacia el exterior (a través del canal serie) la representación ASCII del dato especificado como parámetro. El

CAPÍTULO 4: LENGUAJE ARDUINO

dato en sí puede ser de cualquier tipo (excepto array o estructura): carácter, cadena, número entero, número decimal... pero lo que recibirá el destino es su representación ASCII textual. Así pues, si el dato a enviar fuera (por ejemplo) de tipo entero, especificaríamos `Serial.print(78)`; pero el destino recibiría el texto "78". En este caso, además, se podría especificar un segundo parámetro opcional cuyo valor ha de ser alguna constante predefinida de las siguientes: BIN, HEX o DEC: en el primer caso se enviaría la representación binaria del número (es decir, `Serial.print(78,BIN)`; enviaría el texto "1001110"), en el segundo, la representación hexadecimal (es decir, `Serial.print(78,HEX)`; enviaría el texto "4E"), y en el tercero, la representación decimal (la usada por defecto). En el caso de que el dato a enviar sea un número decimal, el segundo parámetro (también opcional) tendrá entonces un valor entero y servirá para indicar el número de decimales que se desean utilizar en la representación ASCII a enviar (por defecto son dos). En el caso de querer especificar explícitamente un valor textual como dato a enviar (en vez de a través de una variable), recordemos que los caracteres se deberán escribir entre comillas simples y las cadenas entre comillas dobles.

El valor de retorno de `Serial.print()` es un dato de tipo *byte* que vale el número de bytes enviados. En el caso de cadenas de caracteres, este valor de retorno coincide con el número de caracteres enviados. El uso o no en nuestro sketch de este valor devuelto dependerá de nuestras necesidades.

La transmisión de los datos realizada por `Serial.print()` es asíncrona. Eso significa que nuestro sketch pasará a la siguiente instrucción y seguirá ejecutándose sin esperar a que empiece a realizarse el envío de los datos. Si este comportamiento no es el deseado, se puede añadir justo después de `Serial.print()` la instrucción `Serial.flush()` –que no tiene ningún parámetro ni devuelve ningún valor de retorno–, instrucción que espera hasta que la transmisión de los datos sea completa para continuar la ejecución del sketch.

NOTA: En realidad, `Serial.print()` no envía los datos directamente al "exterior", sino que los va almacenando en una memoria intermedia del microcontrolador llamada "buffer de escritura" (o "buffer de salida") desde donde se van "expulsando" hacia afuera al ritmo que sea posible. Si en algún momento este buffer estuviera lleno, el comportamiento de las líneas `Serial.print()` pasaría a ser síncrono (es decir: se parará la ejecución del sketch) hasta que haya hueco en el buffer y así que quepan los datos enviados por esas líneas. Para saber en un determinado momento el número de bytes libres que aún quedan en el buffer (o dicho de otra forma, la cantidad de datos que aún se pueden enviar al canal serie sin parar el sketch) se puede utilizar la instrucción `Serial.availableForWrite()`, la cual no tiene parámetros y devuelve precisamente esa información. Para saber el tamaño total del buffer, se puede escribir esta misma instrucción –antes de realizar ninguna operación `Serial.print()`– dentro de la sección `setup()`.

EL MUNDO GENUINO-ARDUINO

Serial.println(): hace exactamente lo mismo que *Serial.print()*, pero además, añade automáticamente al final de los datos enviados dos caracteres extra: el de retorno de carro (código ASCII nº 13) y el de nueva línea (código ASCII nº 10). La consecuencia es que al final de la ejecución de *Serial.println()* se efectúa un salto de línea. Tiene los mismos parámetros y los mismos valores de retorno que *Serial.print()*

Se puede simular el comportamiento de *Serial.println()* mediante *Serial.print()* si se añaden manualmente estos caracteres. El carácter retorno de carro se representa con el símbolo '\r', y el de salto de línea con '\n'. Así pues, *Serial.print("holo\r\n");* sería equivalente a *Serial.println("holo");*. Otro carácter ASCII no imprimible útil que podemos representar es el tabulador (mediante '\t').

Llegados a este punto, ya hemos visto el significado de todas las líneas de nuestro primer sketch de este capítulo, y por tanto, deberíamos de ser capaces de entender su comportamiento. Analicémoslo, pues. Si recordamos el código, primero declarábamos una variable global de tipo *int* y la inicializábamos con un valor de 555. Seguidamente, arrancábamos la ejecución del programa abriendo el canal serie (a una velocidad de 9600 bits/s) para que la placa pudiera establecer comunicación con nuestro computador. Y finalmente, ya en la sección *void loop()*, primero envíábamos el valor actual de la variable a nuestro computador (que se supone que es el dispositivo conectado vía USB al canal serie abierto). Seguidamente aumentábamos en una unidad ese valor, para justo volver a enviar ese nuevo valor al computador, y volver a aumentar en una unidad su valor, y volverlo a enviar... así infinitamente hasta que la placa dejara de recibir alimentación. Lo que veríamos por el "Serial monitor" sería precisamente esos valores (uno en cada línea diferente porque *Serial.println()* introduce un salto de línea automático) que irían aumentando de uno en uno sin parar. En el momento que se llegara al valor máximo permitido por el tipo de datos de la variable (que en el caso de un *int* es de 32767) se seguiría por el valor mínimo (-32768) y se seguiría aumentando desde allí, en un ciclo sin fin.

Serial.write(): envía a través del canal serie un dato (especificado como parámetro) desde el microcontrolador hacia el exterior sin efectuar ninguna transformación de formato. Este dato a enviar solo puede ocupar un byte, por lo que ha de ser solamente de tipo *char* o *byte* (aunque, en realidad, también es capaz de transmitir cadenas de caracteres porque las trata como una mera secuencia de bytes independientes uno tras otro). En cambio, otros tipos de datos que ocupen más de un byte indisolublemente (como *los int, word, float...*) no serán enviados correctamente. Su valor de retorno es un dato de tipo *byte* que vale precisamente el número de bytes enviados.

NOTA: En realidad, técnicamente sí es posible enviar datos que ocupan más de un byte mediante *Serial.write()*, pero eso requiere manipular individualmente cada uno de los bytes que forman ese dato, algo que no se verá en este libro por ser de un nivel muy avanzado.

La gracia de *Serial.write()* está en que, tal como ya hemos indicado, el dato es enviado siempre directamente sin interpretar. Es decir, se envía como un byte (o una serie de bytes) tal cual, sin representación alguna. Esto no pasa con *Serial.print()*, en la cual se puede jugar con los formatos binario, hexadecimal, etc. Por tanto, esta instrucción está pensada para la transferencia directa de datos con otro dispositivo sin una previsualización por nuestra parte.

Hay que tener en cuenta, no obstante, que si observamos en el "Serial monitor" los datos enviados por *Serial.write()*, veremos que se muestran (tanto si son de tipo *char* como de tipo *byte*) en forma de sus correspondientes caracteres ASCII. Esto es así porque es el propio "Serial monitor" quien realiza en tiempo real esta "traducción" ASCII (no nuestro sketch). El siguiente código ilustra mejor el comportamiento recién descrito:

Ejemplo 4.7

```
char cadena[]="hola";
byte bytesDevueltos;
void setup() {
    Serial.begin(9600);
    bytesDevueltos=Serial.write(cadena);
    Serial.println(bytesDevueltos);
}
void loop() {}
```

Si ejecutamos el código anterior y observamos el "Serial monitor", veremos que aparece el valor "hola4". El 4 final muestra el valor de la variable *bytesDevueltos*, que guarda lo devuelto por *Serial.write(cadena);*, confirmando así que se han enviado por el canal serie 4 bytes, los correspondientes precisamente a la cadena "hola".

Existe otra manera de utilizar *Serial.write()* que nos permite enviar una tira de datos de tipo *byte* (o *char*) de una sola vez. Para ello, esta instrucción deberá tener dos parámetros: el nombre del array (obligatoriamente de tipo *byte* o *char*) que contendrá esos datos a enviar y el número de elementos de ese array (empezando siempre por el primero) que se quiere realmente transmitir. Este último valor no tiene por qué coincidir con el total de elementos del array (fácilmente calculable mediante *sizeof()*), sino que puede ser una cantidad de elementos menor. Por otro lado, también sería posible enviar, en vez de un array, una estructura (siempre que, eso sí,

EL MUNDO GENUINO-ARDUINO

todos sus elementos sean igualmente de tipo *byte* o *char*) indicando como primer parámetro el nombre de dicha estructura precedido (¡ojo!) de la expresión de casting especial *(byte*)&* o *(char*)&* –según el tipo de sus elementos– y, como segundo parámetro, el número de elementos que se quieren realmente transmitir. El siguiente código ilustra mejor estas opciones:

Ejemplo 4.8

```
//El array ha de ser de tipo 'byte' (o 'char')
byte arrayBytes[ ]={65,66,67,68}; //El carácter ASCII correspondiente al nº65 es 'A', al nº66 es 'B',...
//Los elementos del struct han de ser todos de tipo 'byte' (o 'char')
struct s {byte num; byte letra;};
s structBytes = { 65, 'A' };
void setup() {
    Serial.begin(9600);
    //Envio todos los elementos actuales del array (el "Serial monitor" mostrará los caracteres ASCII)
    Serial.write(arrayBytes,sizeof(arrayBytes));
    Serial.println();
    //Envio todos los elementos del struct (el "Serial monitor" mostrará los caracteres ASCII)
    Serial.write((byte*)&structBytes, sizeof(structBytes));
}
void loop() {}
```

Sea como sea, la transmisión del byte es asíncrona. Eso significa que nuestro sketch pasará a la siguiente instrucción para seguir ejecutándose sin esperar a que empiece a realizarse el envío de ese/os byte/s. Si este comportamiento no es el deseado, se puede añadir justo después de *Serial.write()* la instrucción **Serial.flush()** –que no tiene ningún parámetro y no devuelve ningún valor de retorno–, la cual esperará hasta que la transmisión de los datos sea completa para continuar la ejecución del sketch.

Uso del "Serial Plotter"

El IDE Arduino ofrece una funcionalidad que nos puede ser muy útil en determinadas ocasiones: el "Serial Plotter". Esta herramienta (disponible a través de la opción del menú "Tools->"Serial Plotter") muestra una gráfica en tiempo real de los valores (numéricos, se entiende) que nuestra placa Arduino va enviando al exterior mediante *Serial.println()*. Gracias al "Serial Plotter", pues, podemos hacer un seguimiento de dichos valores de tipo visual (y, por tanto, más intuitivo y agradable) que si usáramos el "Serial Monitor". La utilidad práctica más inmediata de ello es el poder observar de forma rápida y directa la tendencia existente en los datos recibidos a través de un sensor analógico.

Es importante tener en cuenta que "Serial Plotter" solamente reconoce como valores aptos para representar los que van seguidos de un salto de línea. Esto significa que solamente podemos utilizar *Serial.println()* para enviarle los datos a visualizar (es decir, *Serial.write()* o *Serial.print()* a secas no funcionan). Más en concreto, lo que hace "Serial Plotter" internamente es separar la cadena recibida por saltos de línea (identificando cada trozo como un dato diferente), eliminar los posibles espacios en blanco que pudieran existir antes o después de cada dato en sí y convertirlo (si es posible) en un valor de tipo *double* para mostrarlo en la gráfica de forma numérica.

Lo explicado en el párrafo anterior tiene como consecuencia práctica que si deseáramos enviar al "Serial Plotter" varios datos a la vez, cada uno correspondiente (por ejemplo) a un sensor diferente, deberíamos enviarlos todos en una misma orden *Serial.println()* pero separándolos entre sí mediante comas, espacios en blanco o tabuladores (es decir, así: *Serial.println("3.4, 5.4, 1.0");* . Al hacer esto (esto es, enviar varios valores numéricos a la vez) se generará una gráfica independiente por cada uno, de manera que si se respeta el mismo orden de los valores en cada invocación a *Serial.println()*, obtendremos una evolución gráfica y en tiempo real de cada entrada por separado.

Instrucciones para recibir datos desde el exterior

Hasta ahora hemos visto instrucciones que permiten al microcontrolador enviar datos a su entorno. Pero ¿cómo se hace a la inversa? Es decir: ¿cómo se pueden enviar datos al microcontrolador (para que este los recoja y los procese) que provengan de su entorno, como por ejemplo nuestro computador?

Desde el "Serial monitor" enviar datos a la placa es muy sencillo: no hay más que escribir lo que queramos en la caja de texto allí mostrada y pulsar el botón "Send". No obstante, si el sketch que se está ejecutando en la placa no está preparado para recibir y procesar estos datos, esa transmisión no llegará a ningún sitio. Por tanto, necesitamos recibir convenientemente en nuestros sketches los datos que lleguen a la placa vía comunicación serie. Para ello, disponemos de dos instrucciones básicas: *Serial.available()* y *Serial.read()*.

Serial.available(): devuelve el número de bytes –caracteres– disponibles para ser leídos que provienen del exterior a través del canal serie (vía USB o vía pines TX/RX). Estos bytes ya han llegado al microcontrolador y permanecen almacenados temporalmente en una pequeña memoria de 64 bytes que tiene el chip TTL-UART –llamada "buffer de lectura" o "buffer de entrada"–

EL MUNDO GENUINO-ARDUINO

hasta que sean procesados mediante la instrucción *Serial.read()* o similares. Si no hay bytes para leer, esta instrucción devolverá 0. No tiene parámetros.

Serial.read(): devuelve el primer byte aún no leído de los que estén almacenados en el buffer de entrada del chip TTL-UART. Al hacerlo, lo elimina de ese buffer. Para devolver (leer) el siguiente byte, se ha de volver a ejecutar *Serial.read()*. Y hacer así hasta que se hayan leído todos. Cuando no haya más bytes disponibles, *Serial.read()* devolverá -1. No tiene parámetros.

Veamos un código básico de estas nuevas instrucciones:

Ejemplo 4.9

```
byte byteRecibido = 0;
void setup() {
    Serial.begin(9600);
}
void loop() {
    if (Serial.available() > 0) {
        byteRecibido = Serial.read();
        Serial.write("Byte recibido: ");
        Serial.write(byteRecibido);
    }
}
```

En el código anterior hemos introducido un elemento del lenguaje que todavía no hemos visto: el condicional *if*. Lo explicaremos en profundidad en su apartado correspondiente de este capítulo, pero baste por ahora saber que un *if* mira si la condición escrita entre paréntesis es cierta o no: si lo es, se ejecutan las instrucciones escritas dentro de sus llaves, y si no, no. La condición vemos que es *Serial.available() > 0*, así que lo que se está comprobando es si existen o no datos almacenados en el buffer de entrada del chip TTL-UART. Si esta condición es cierta, se ejecuta el interior del *if*, que lo que hace básicamente es leer el primer byte disponible que haya en el buffer (eliminándolo de allí) y enviarlo al "Serial monitor". Como todo esto ocurre dentro de la sección *void loop()*, seguidamente regresaremos otra vez a su principio para volver a comprobar si aún existen datos almacenados en el buffer de entrada. Si sigue siendo así, se leerá el siguiente byte disponible y se volverá a enviar al "Serial monitor" (y este lo mostrará justo después del anterior byte, sin separación de ningún tipo entre ambos). Seguidamente se comprobará otra vez si sigue habiendo datos en el buffer, en cuyo caso se leerá el siguiente byte. Y así hasta que ya se hayan leído todos los bytes disponibles. En ese momento, la condición del *if* pasará a ser falsa (porque *Serial.available()* devolverá 0) y por tanto

void loop() no ejecutará nada. Pero como a cada repetición de *void loop()* se seguirá comprobando si hay o no datos en el buffer, en el momento que vuelva a haber, la condición del *if* volverá a ser cierta y por tanto otra vez se empezarán a leer los bytes disponibles allí, uno a uno.

Un detalle muy importante del código anterior es que, tal como se puede observar, el valor devuelto por *Serial.read()* se guarda en una variable de tipo *byte*. Esto implica que dicho valor será interpretado en nuestro sketch como un dato de tipo numérico. Es decir: si ejecutáramos el código anterior y, por ejemplo, desde el "Serial monitor" enviáramos la letra 'a', *Serial.read()* la recibiría pero lo que guardaría en la variable de tipo *byte* sería el valor numérico de esa letra en la tabla ASCII (en este caso, 97); igualmente, si *Serial.read()* recibiera, por ejemplo, el valor '1', en realidad lo que guardaría en la variable de tipo *byte* sería el valor numérico 49, y así. Esto es fácilmente observable si sustituimos en el código anterior la línea *Serial.write(byteRecibido);* por la línea *Serial.print(byteRecibido);*: al hacer este cambio, el "Serial monitor" no realiza ninguna interpretación por nosotros, mostrando el valor numérico "real" de la variable. Sin embargo, si el tipo de la variable utilizado para guardar el valor devuelto por *Serial.read()* fuera *char* (recordemos que es el otro tipo de datos que ocupa 1 byte en memoria), lo que ocurriría es que 'a' entonces sí sería interpretado dentro de nuestro sketch como carácter (asimismo, '1' sería tratado como el carácter '1', etc.). La consecuencia de todo ello es que debemos pensar previamente qué utilidad le vamos a dar en nuestro sketch al valor devuelto (¿número o carácter?) para entonces decidir el tipo de datos de la variable que lo guardará.

El siguiente código muestra otro ejemplo de uso combinado de *Serial.read()* y *Serial.write()* –y donde se vuelve a utilizar el condicional *if*–. Concretamente, se encarga de leer continuamente cada byte recibido por el canal serie (introducidos, por ejemplo, en forma de caracteres mediante la caja de texto del "Serial monitor") y de reenviarlo de nuevo (a modo de eco) a dicho canal, excepto cuando detecte que el valor del byte procesado en ese momento es 44 (el cual corresponde al valor ASCII del carácter coma ','): en ese caso, lo que reenviará al canal serie será otro byte (directamente el carácter arroba '@'):.

Ejemplo 4.10

```
byte byteRecibido = 0;
void setup() {
    Serial.begin(9600);
}
void loop() {
```

EL MUNDO GENUINO-ARDUINO

```
if (Serial.available() > 0) {  
    byteRecibido = Serial.read();  
    //Miro si el valor del byte recibido es igual a 44 (correspondiente al carácter ASCII ',')  
    //Es importante fijarse que en la condición del 'if' hay escritos dos signos '=' seguidos  
    if(byteRecibido==44){  
        Serial.write("@");  
    }  
    //Si, en cambio, el byte recibido no es igual a 44...(notar que 'no igual' se escribe '!=')  
    if(byteRecibido!=44){  
        Serial.write(byteRecibido);  
    }  
}
```

Existen otras instrucciones además de *Serial.read()* que leen datos del buffer de entrada del chip TTL-UART de formas más específicas, las cuales nos pueden venir bien en determinadas circunstancias. Por ejemplo, *Serial.peek()*:

Serial.peek(): devuelve el primer byte aún no leído de los que estén almacenados en el buffer de entrada. No obstante, a diferencia de *Serial.read()*, ese byte leído no se borra del buffer, con lo que las próximas veces que se ejecute *Serial.peek()* –o una vez *Serial.read()*– se volverá a leer el mismo byte. Si no hay bytes disponibles para leer, *Serial.peek()* devolverá -1. Esta instrucción no tiene parámetros.

Pero las más útiles son, sobre todo, *Serial.parseInt()*/*Serial.parseFloat()*, *Serial.readBytes()*/*Serial.readBytesUntil()* y *Serial.readString()*/*Serial.readStringUntil()*:

Serial.parseInt(): lee del buffer de entrada (eliminándolos de allí) todos los datos hasta que se encuentre con un número entero (o bien, si esto no ocurre, hasta que se haya superado un determinado tiempo de espera, el cual por defecto es de 1 segundo, tiempo que puede ser modificado previamente mediante la instrucción *Serial.setTimeout()*). Si sí se encuentra un número entero, al detectar el primer carácter posterior no válido, *Serial.parseInt()* dejará de leer (y por tanto, no seguirá eliminando datos del buffer). Su valor de retorno –de tipo *long*– será el número entero encontrado (o bien 0 si no se ha encontrado ninguno en el tiempo de espera configurado). Esta instrucción no tiene parámetros.

Serial.parseFloat(): lee del buffer de entrada (eliminándolos de allí) todos los datos hasta que se encuentre con un número decimal (o bien, si esto no

ocurre, hasta que se haya superado un determinado tiempo de espera, el cual por defecto es de 1 segundo, tiempo que puede ser modificado previamente mediante la instrucción `Serial.setTimeout()`. Si sí se encuentra un número decimal, al detectar el primer carácter posterior no válido, `Serial.parseFloat()` dejará de leer (y por tanto, no seguirá eliminando datos del buffer). Su valor de retorno –de tipo `float`– será entonces ese número decimal encontrado (o bien 0.00 si no se ha encontrado ninguno en el tiempo de espera configurado). Esta instrucción no tiene parámetros.

Serial.setTimeout(): tiene un parámetro (de tipo `long`) que sirve para establecer el número de milisegundos máximo que las instrucciones `Serial.parseInt()` y `Serial.parseFloat()` –y también `Serial.readBytesUntil()`, `Serial.readBytes()`, `Serial.readStringUntil()` y `Serial.readString()`– esperarán a la llegada de datos al búfer de entrada serie. Si alguna de estas instrucciones no recibe suficientes datos y se supera ese tiempo, el sketch continuará su ejecución en la línea siguiente. El tiempo de espera por defecto es de 1000 milisegundos. Esta instrucción se suele escribir en `void setup()`. No tiene valor de retorno.

El siguiente código (junto con la ayuda del botón "Send" del "Serial monitor") nos permite probar el uso de `Serial.parseFloat()`:

Ejemplo 4.11

```
float numero;
void setup(){
    Serial.begin(9600);
}
void loop(){
    //Vacía el buffer hasta reconocer algún número decimal (si se recibe antes de 1s de espera)
    numero=Serial.parseFloat();
    /*Imprime el doble del número decimal detectado (con cuatro cifras decimales).
     Si no se ha encontrado ninguno, imprime 0.00 */
    Serial.println(numero*2,4);
    /*Lee un byte más y lo imprime. Si se hubiera detectado un número decimal, ese byte sería el
     carácter que está justo después de él en el buffer. Si el buffer estuviera vacío, se devolverá -1 */
    Serial.println(Serial.read());
}
```

Serial.readBytes(): lee del buffer de entrada (eliminándolos de allí y almacenándolos en un array –de tipo `char[]`– especificado como primer parámetro) la cantidad de bytes especificada como segundo parámetro (o

EL MUNDO GENUINO-ARDUINO

bien, si no llegan suficientes bytes, hasta que se haya superado el tiempo especificado por `Serial.setTimeout()`). En cualquier caso, mientras esta función permanece a la espera de leer bytes, el sketch permanece bloqueado. Esta instrucción devuelve el número de bytes leídos del buffer (por lo que un valor 0 significa que no se encontraron datos válidos).

Serial.readBytesUntil(): lee del buffer de entrada (eliminándolos de allí y almacenándolos en un array –de tipo `char[]`– especificado como segundo parámetro) la cantidad de bytes especificada como tercer parámetro, o bien, si dicha cantidad aún no se ha alcanzado, hasta recibir la cadena de caracteres –o carácter individual– especificada como primer parámetro (que hace, por tanto, de marca de final de lectura). Si después de transcurrir el tiempo especificado por `Serial.setTimeout()` aún no hubieran llegado suficientes bytes o no se hubiera recibido la marca de final, esta instrucción igualmente dejará de leer y la ejecución del sketch proseguirá en la siguiente línea. Esta instrucción devuelve el número de bytes leídos del buffer (por lo que un valor 0 significa que no se encontraron datos válidos).

El código siguiente muestra el uso de `Serial.readBytes()`. Concretamente, primero obtiene de 20 en 20 los bytes que haya en el buffer de entrada y los almacena en el array `miarray`, mostrando seguidamente la cantidad de bytes obtenidos y sus valores en forma de cadena. Si en el buffer hay más de 20 bytes, a la siguiente repetición de `void loop()` se volverá a ejecutar `Serial.readBytes()`, con lo que leerá los siguientes 20 bytes del buffer y se sobrescribirán los valores que había en el array por los nuevos. Se puede utilizar el botón "Send" del "Serial monitor" para enviar caracteres a la placa y observar el resultado:

Ejemplo 4.12

```
char miarray[30];
byte bytesleidos;
void setup(){
    Serial.begin(9600);
}
void loop(){
    bytesleidos=Serial.readBytes(miarray,20);
    Serial.println(bytesleidos);
    Serial.println(miarray);
}
```

Serial.readString(): lee (y a la vez, elimina) los bytes del buffer de entrada durante el tiempo "de lectura" especificado por `Serial.setTimeout()` y los

añade (interpretándolos como caracteres) a una cadena, la cual, una vez superado ese tiempo de lectura, será el valor devuelto, en forma de dato de tipo *String*. Hay que tener en cuenta que hasta que no haya transcurrido completamente el tiempo de lectura, el sketch permanecerá bloqueado (en otras palabras, solamente tras finalizar dicho tiempo la ejecución del sketch proseguirá en la siguiente línea). Esta instrucción no tiene parámetros.

Serial.readStringUntil(): lee (y a la vez, elimina) los bytes del buffer de entrada interpretándolos como caracteres de una cadena. Esta lectura la realizará hasta recibir una determinada cadena de caracteres –o carácter individual– que hará de marca de final de lectura y que se indicará como único parámetro de la instrucción. Esta instrucción dejará de leer igualmente si después de transcurrir el tiempo especificado por *Serial.setTimeout()* aún no se ha recibido dicha marca de final. En cualquier caso, la ejecución del sketch proseguirá por su siguiente línea y el valor devuelto por esta instrucción será la cadena de caracteres leída (en forma de dato de tipo *String*).

El código siguiente muestra un ejemplo de uso de *Serial.readStringUntil()*. Si lo ponemos en marcha y abrimos el "Serial monitor" (u otro terminal serie cualquiera), podremos observar que primero nos avisa de que la placa Arduino empezará a leer posibles caracteres llegados a través del canal serie durante los próximos 5 segundos. En el transcurso de ese tiempo, el sketch permanecerá parado recibiendo los datos del exterior. Una vez alcanzados esos 5 segundos, se guardarán los caracteres recibidos en la variable *cad*, se informará del fin de la lectura, se mostrará el valor de *cad* y (debido a la naturaleza repetitiva de la sección *void loop()*) se volverá a leer el contenido llegado del canal serie otros 5 segundos más, y así sucesivamente. Lo interesante es que podemos detener la lectura en cualquier momento antes de llegar a los 5 segundos si el carácter recibido resulta ser el carácter-marcador de fin de lectura, que en este caso resulta ser los dos puntos (':'). Es más, si se introdujera en el "Serial monitor" una cadena tal como por ejemplo "ab:cd:ef", al pulsar en el botón "Send" veríamos que la lectura de dicha cadena se realiza por trozos, al detectarse el carácter-marcador de fin de lectura en varios sitios sucesivos (es decir, primero se leerá "ab", después "cd" y finalmente "ef", todo ello de forma inmediata):

Ejemplo 4.13

```
String cad;
void setup() {
    Serial.begin(9600);
```

EL MUNDO GENUINO-ARDUINO

```
//Aumento el tiempo de lectura a 5s
Serial.setTimeout(5000);
}

void loop() {
    Serial.println("Empiezo a leer:");
    //Leo el canal serie durante 5s (o hasta leer la marca ':') y guardo en 'cad' los caracteres recibidos
    cad = Serial.readStringUntil(':');
    Serial.println("He acabado de leer");
    //Muestro la cadena leída hasta ahora
    Serial.println(cad);
}
```

Finalmente, también tenemos las instrucciones *Serial.find()*/*Serial.findUntil()*:

Serial.find(): lee datos del buffer de entrada (eliminándolos de allí) hasta que se localice la cadena de caracteres –o un carácter individual– a buscar (indicada/o como único parámetro) o bien se supere un cierto tiempo de espera, predefinido de 1 segundo pero configurable mediante *Serial.setTimeout()*... lo que ocurra antes. En cualquier caso, mientras esta función permanece a la espera de encontrar el valor deseado, el sketch permanece bloqueado. La instrucción devuelve *true* si se localiza la cadena deseada antes de finalizar la búsqueda o *false* si no.

Serial.findUntil(): lee datos del buffer de entrada (eliminándolos de allí) hasta que se localice la cadena de caracteres –o un carácter individual– a buscar (indicada/o como primer parámetro) o bien se detecte una marca de final de búsqueda (representada en forma de otra cadena –o carácter– indicada como segundo parámetro), o bien se supere un cierto tiempo de espera, predefinido de 1 segundo pero configurable mediante *Serial.setTimeout()*... lo que ocurra antes. En cualquier caso, mientras esta función permanece a la espera de encontrar el valor deseado, el sketch permanece bloqueado. La instrucción devuelve *true* si se localiza la cadena deseada antes de finalizar la búsqueda o *false* si no.

El código siguiente utiliza *Serial.find()* para leer continuamente los datos presentes en el buffer de entrada en busca de la palabra "hola". Se puede probar su comportamiento escribiendo en la caja de texto del "Serial monitor" las cadenas que deseemos enviar a la placa. Si la cadena "hola" se encuentra, se mostrará por el "Serial monitor" la palabra "Encontrado":

Ejemplo 4.14

```

boolean encontrado;
void setup(){
    Serial.begin(9600);
}
void loop(){
    encontrado=Serial.find("hola");
    if (encontrado == true){
        Serial.println("Encontrado");
    }
}

```

Aquí volvemos a ver un ejemplo de *if*: básicamente lo que comprueba es que el valor devuelto por *Serial.find()* sea verdadero para así poder imprimir "Encontrado". Es importante insistir en que se han de escribir los dos iguales en la condición del *if* (ya hablaremos de ello en el apartado correspondiente).

Es posible recibir un conjunto de bytes de forma no bloqueante (es decir, no tener que pausar nuestro sketch un determinado "timeout" mediante *Serial.readBytes()*, *Serial.readString()* o similares, sino poder ir ejecutándolo a la vez que fuera recibiendo los eventuales datos desde el canal serie) pero para ello necesitaremos emplear técnicas de programación relativamente avanzadas que no veremos hasta el capítulo 6, cuando intentemos gestionar en tiempo real (es decir, sin pausas y a la vez) diferentes entradas y salidas de nuestra placa a través de órdenes enviadas puntualmente a ella a través del canal serie. Remito al lector a dicho capítulo para profundizar en este aspecto, pues.

Los objetos serie de otras placas Arduino diferentes de la UNO

Hasta ahora hemos supuesto el uso de la placa Arduino UNO, la cual tiene solo un chip TTL-UART (o mejor dicho, el microcontrolador en el que está basada) y permite por tanto un único objeto serie, llamado *Serial*. No obstante, otros modelos de placa Arduino disponen de más objetos de este tipo, y por tanto, de una mayor versatilidad en el uso de la comunicación serie.

Por ejemplo, la placa Arduino Mega ofrece cuatro chips TTL-UART. Esto significa que podemos utilizar hasta cuatro objetos serie, llamados *Serial*, *Serial1*, *Serial2* y *Serial3*. Al igual que ocurría en el modelo UNO, el objeto *Serial* está asociado a los pines 0 (RX) y 1 (TX) y es el único objeto serie que está asociado además a la conexión USB. Por otro lado, el objeto *Serial1* está asociado a los pines 18 (TX) y 19 (RX), el *Serial2* a los pines 16 (TX) y 17 (RX) y el *Serial3* a los pines 14 (TX) y 15 (RX).

EL MUNDO GENUINO-ARDUINO

Cada uno de estos objetos se puede abrir independientemente (escribiendo `Serial.begin(9600); Serial1.begin(9600); Serial2.begin(9600); o Serial3.begin(9600);` respectivamente), y pueden enviar y recibir datos también independientemente.

Las placas Arduino Yún y Micro solo disponen, aparte del objeto *Serial*, del objeto *Serial1*. En este caso es para separar las dos posibles vías de comunicación serie que puede manejar el nativamente microcontrolador ATmega32U4: el objeto *Serial* se reserva para la comunicación USB-ACM (que es diferente de la comunicación USB usada para las simulaciones de teclado y ratón) y el objeto *Serial1* se reserva para la transmisión de información a través de los pines 0 (RX) y 1 (TX). No obstante, en el caso particular de la placa Yún, estos pines 0 y 1 (y, por tanto, el objeto *Serial1* asociado) están conectados al microprocesador AR9331 (de hecho, estos pines son el canal de comunicación existente entre ambos chips), por lo que no podrán ser conectados a dispositivos externos.

La placa Arduino Due en este sentido es similar a la placa Mega: al igual que ella dispone de cuatro puertos serie manipulables mediante los nombres de *Serial*, *Serial1*, *Serial2* y *Serial3*. E igualmente, el objeto *Serial*, además de estar conectado a los pines 0 y 1 lo está, a través del chip ATmega16U2, al "puerto USB de programación" de la placa, por lo que ese será el objeto que deberemos utilizar en nuestros sketches para enviar y recibir datos serie a través de un cable USB conectado a ese zócalo. Asimismo, el objeto *Serial1* está asociado también a los pines 18 (TX) y 19 (RX), el *Serial2* a los pines 16 (TX) y 17 (RX) y el *Serial3* a los pines 14 (TX) y 15 (RX). La mayor novedad aparece, no obstante, si quisieramos enviar y recibir datos serie directamente al chip SAM3X a través del "puerto USB nativo" que también ofrece la placa, porque en este caso el objeto a utilizar en nuestros sketches debería de ser uno diferente llamado *SerialUSB*.

La placa Arduino Zero, aunque por hardware sería capaz de ofrecer más puertos TTL-UART libres (de hecho, dispone de unos puertos especiales llamados "SERCOM" que podrían configurarse a la carta como puertos extra TTL-UART, SPI, I²C, etc.), por ahora solamente permite en nuestro sketches los mismos dos objetos que hay disponibles en las placas Yún o Micro, es decir: un objeto llamado *Serial* reservado para la transmisión serie realizada a través del chip EDBG (es decir, vía "puerto USB de programación") y otro objeto llamado *Serial1* reservado para la transmisión de información a través de sus pines nº 0 (RX) y nº 1 (TX). También podríamos enviar y recibir datos directamente al chip SAM-D21 a través del "puerto USB nativo", pero en este caso (de forma similar a lo que ocurría con la placa Arduino Due) el objeto a utilizar en nuestros sketches debería de ser otro llamado *SerialUSB*.

Como información curiosa, no está de más saber que si estamos cansados de usar el nombre *Serial* (o si tenemos varios puertos serie disponibles –*Serial1*, *Serial2*,...– como en el caso de usar las placas Arduino Mega/Yún, etc.) y queremos usar nombres más descriptivos, puede hacerse de la siguiente forma (en el siguiente ejemplo hemos supuesto que tenemos dos dispositivos –un módulo bluetooth y otro wifi– conectados a dos puertos serie diferentes, además del puerto USB-ACM):

```
HardwareSerial &pc    = Serial;      //Asigno el nombre de "pc" a la conexión Serial
HardwareSerial &bluetooth = Serial1; //Asigno el nombre de "bluetooth" a la conexión Serial1
HardwareSerial &wifi    = Serial2;    //Asigno el nombre de "wifi" a la conexión Serial2
void setup(){
    pc.begin(9600);           bluetooth.begin(115200);   wifi.begin(9600);
}
void loop() {
    pc.println(millis()/1000); bluetooth.write(3);       wifi.println("Hola");
    delay(1000);
}
```

INSTRUCCIONES DE GESTIÓN DEL TIEMPO

Estas instrucciones no pertenecen a ningún objeto, así que se escriben directamente:

millis(): devuelve el número de milisegundos (ms) desde que la placa Arduino empezó a ejecutar el sketch actual. Este número se reseteará a cero aproximadamente después de 50 días (cuando su valor supere el máximo permitido por su tipo, que es *unsigned long*). No tiene parámetros.

micros(): devuelve el número de microsegundos (μ s) desde que la placa Arduino empezó a ejecutar el sketch actual. Este número –de tipo *unsigned long*– se reseteará a cero aproximadamente después de 70 minutos. Esta instrucción tiene una resolución de 4 μ s (es decir, que el valor returned es siempre un múltiplo de cuatro). Recordar que 1000 μ s es un milisegundo y por tanto, 1000000 μ s es un segundo. No tiene parámetros.

delay(): pausa el sketch durante la cantidad de milisegundos especificados como parámetro –de tipo *unsigned long*–. Es importante tener en cuenta que mientras dure la ejecución de esta instrucción, la placa Arduino no podrá hacer nada más (ni recibir datos de entradas, ni ordenar acciones a salidas, etc.), por lo que se recomienda usarla con precaución. No tiene valor de retorno.

EL MUNDO GENUINO-ARDUINO

delayMicroseconds(): pausa el sketch durante la cantidad de microsegundos especificados como parámetro –de tipo *unsigned long*– . Al igual que ocurre con *delay()*, la placa Arduino se mantiene totalmente "ocupada" en mantener la parada. Actualmente el máximo valor que se puede utilizar con precisión es de 16383. Para esperas mayores que esta, se recomienda usar la instrucción *delay()*. El mínimo valor que se puede utilizar con precisión es de 3 μ s. No tiene valor de retorno.

Un código sencillo de alguna de las instrucciones anteriores es este:

Ejemplo 4.15

```
unsigned long time;  
void setup(){ Serial.begin(9600); } //Los saltos de línea son prescindibles en el código de los sketches  
void loop(){  
    time = micros();  
    Serial.println(time);  
    delay(1000);  
}
```

Si se ejecuta el código anterior se puede ver por el "Serial monitor" cómo va aumentando el tiempo que pasa desde que se puso en marcha el sketch. El valor observado va aumentando aproximadamente un segundo cada vez. Otro código ilustrativo es el siguiente:

Ejemplo 4.16

```
unsigned long inicio, fin, transcurrido;  
void setup(){ Serial.begin(9600); }  
void loop(){  
    inicio=millis();  
    delay(1000);  
    fin=millis();  
    transcurrido=fin-inicio;  
    Serial.print(transcurrido);  
    delay(500);  
}
```

En el código anterior se puede ver una manera de contar el tiempo transcurrido entre dos momentos determinados. El procedimiento es guardar en una variable el valor devuelto por *millis()* en el momento inicial, y guardar en otra variable diferente el valor devuelto por *millis()* en el momento final, para seguidamente restar uno del otro y averiguar así el lapso de tiempo transcurrido (que en el ejemplo debería de ser de aproximadamente un segundo). Este cálculo en el código anterior se realiza cada medio segundo.

INSTRUCCIONES MATEMÁTICAS, TRIGONOMÉTRICAS Y DE PSEUDOALEATORIEDAD

El lenguaje Arduino dispone de una serie de instrucciones matemáticas y de pseudoaleatoriedad que nos pueden venir bien en nuestros proyectos. Son estas:

abs(): devuelve el valor absoluto de un número pasado por parámetro (el cual puede ser tanto entero como decimal). Es decir, si ese número es positivo (o 0), lo devuelve sin alterar su valor; si es negativo, lo devuelve "convertido en positivo". Por ejemplo, 3 es el valor absoluto tanto de 3 como de -3.

min(): devuelve el mínimo de dos números pasados por parámetros (los cuales pueden ser tanto enteros como decimales).

max(): devuelve el máximo de dos números pasados por parámetros (los cuales pueden ser tanto enteros como decimales).

Muchas veces se utilizan las funciones *min()* y *max()* para restringir el valor mínimo o máximo que puede tener una variable cuyo valor proviene de un sensor. Por ejemplo, si tenemos una variable llamada *sensVal* que obtiene su valor de un sensor, la línea *sensVal = min(sensVal, 100);* se asegura que dicha variable no va a tener nunca un valor más pequeño de 100, a pesar de que ese sensor reciba en un momento dado valores más pequeños. En este sentido, nos puede ser útil otra instrucción más específica llamada *constrain()*, la cual sirve para contener un valor determinado entre dos extremos mínimo y máximo:

constrain(): recalcula el valor pasado como primer parámetro (llamémosle *x*) dependiendo de si está dentro o fuera del rango delimitado por los valores pasados como segundo y tercer parámetro (llamémoslos *a* y *b* respectivamente, donde *a* siempre ha de ser menor que *b*). Los tres parámetros pueden ser tanto enteros como decimales. En otras palabras:

Si *x* está entre *a* y *b*, *constrain()* devolverá *x* sin modificar

Si *x* es menor que *a*, *constrain()* devolverá *a*

Si *x* es mayor que *b*, *constrain()* devolverá *b*

Una instrucción algo más compleja que las anteriores (pero más versátil) es la instrucción *map()*. Esta instrucción es utilizada en multitud de proyectos para adecuar las señales de entrada obtenidas por diferentes sensores a un rango numérico óptimo para trabajar. Veremos varios ejemplos de su uso en posteriores apartados.

EL MUNDO GENUINO-ARDUINO

map(): modifica un valor –especificado como primer parámetro– el cual inicialmente está dentro de un rango (delimitado con su mínimo –segundo parámetro– y su máximo –tercer parámetro–) para que esté dentro de otro rango (con otro mínimo –cuarto parámetro– y otro máximo –quinto parámetro–) de forma que la transformación del valor sea lo más proporcional posible. Esto es lo que se llama "mapear" un valor: los mínimos y los máximos del rango cambian y por tanto los valores intermedios se adecúan a ese cambio. Todos los parámetros son de tipo *long*, por lo que se admiten también números enteros negativos, pero no números decimales: si aparece alguno en los cálculos internos de la instrucción, este será truncado. El valor devuelto por esta instrucción es precisamente el valor mapeado, también de tipo *long*. El siguiente código muestra un ejemplo de uso:

Ejemplo 4.17

```
void setup(){
    Serial.begin(9600);
    Serial.println(map(0,0,100,200,400));
    Serial.println(map(25,0,100,200,400));
    Serial.println(map(50,0,100,200,400));
    Serial.println(map(75,0,100,200,400));
    Serial.println(map(100,0,100,200,400));
    //El valor puede estar fuera de los rangos, pero igualmente se mapea de la forma conveniente
    Serial.println(map(500,0,100,200,400));
}
void loop(){}  
}
```

Si se observa la salida en el "Serial monitor", se puede comprobar cómo el valor mínimo del rango inicial (0) se mapea al valor mínimo del rango final (200), que el valor 25 (una cuarta parte del rango total inicial) se mapea al valor 250 (una cuarta parte del rango total final), que el valor 50 (la mitad del rango total inicial) se mapea al valor 300 (la mitad del rango total final), que el valor 75 (tres cuartas partes del rango total inicial) se mapea al valor 350 (tres cuartas partes del rango total final), que el valor máximo del rango inicial (100) se mapea al valor máximo del rango final (400) y que un valor fuera del rango inicial (500) se mapea proporcionalmente a otro valor fuera del rango final (1200).

Observar en este último caso que los valores mapeados no se restringen dentro del nuevo rango porque a veces puede ser útil trabajar con valores fuera de rango. Si se desea acotarlos, se debe usar la instrucción *constrain()* o antes o después de *map()*.

Ver también que los mínimos de los dos rangos (el actual y el modificado) pueden tener en realidad un valor mayor que los máximos. De esta forma, se pueden revertir rangos de números. Por ejemplo, `map(x, 1, 50, 50, 1);` invertiría los valores originales: el 1 pasaría a ser el 50, el 2 el 49... y el 50 pasaría a ser el 1, el 49 el 2...

Para los curiosos: matemáticamente, la instrucción `map()` realiza este cálculo: $xmapeado = (x - minactual) * (maxfinal - minfinal) / (maxactual - minactual) + minfinal$ donde x es el valor a mapear, $minactual$ es el valor mínimo del rango actual, $maxactual$ es el valor máximo del rango actual, $minfinal$ es el valor mínimo del rango final y $maxfinal$ es el valor máximo del rango final. De hecho, podríamos implementar esta misma fórmula utilizando variables de tipo `float` para así realizar "manualmente" un mapeado calculado con valores decimales.

El lenguaje Arduino también dispone de instrucciones relacionadas con potencias:

pow(): devuelve el valor resultante de elevar el número pasado como primer parámetro (la "base") al número pasado como segundo parámetro (el "exponente", el cual puede ser incluso una fracción). Por ejemplo, si ejecutamos `resultado = pow(2,5);` la variable `resultado` valdrá 32 (2^5). Ambos parámetros son de tipo `float`, y el valor devuelto es de tipo `double`.

Si se quiere calcular el cuadrado de un número (es decir, el resultado de multiplicar ese número por sí mismo, además de utilizar la instrucción `pow()` poniendo como exponente el número 2, se puede utilizar una instrucción específica que es **sq()**), cuyo único parámetro es el número (de cualquier tipo) que se desea elevar al cuadrado y cuyo resultado se devuelve en forma de número de tipo `double`.

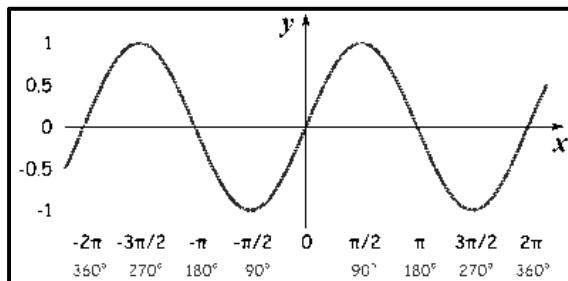
sqrt(): devuelve la raíz cuadrada del número pasado como parámetro (que puede ser tanto entero como decimal). El valor devuelto es de tipo `double`.

También podemos utilizar instrucciones trigonométricas. Va más allá de los objetivos de este libro explicar la utilidad y el significado matemático de estas funciones; si el lector desea profundizar en sus conocimientos de trigonometría, recomiendo consultar el artículo de la Wikipedia titulado "Función trigonométrica".

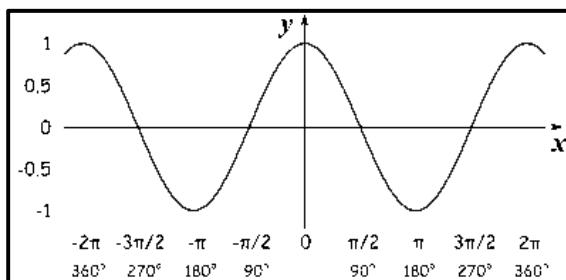
sin(): devuelve el seno de un ángulo, especificado como único parámetro –de tipo `float`–. Este ángulo ha de venir expresado en radianes, por lo que sus valores posibles (mostrados en el eje "x" horizontal de la gráfica siguiente) a menudo estarán dentro del rango entre 0 y 2π ; si estuvieran fuera de él, al

EL MUNDO GENUINO-ARDUINO

ser esta función periódica, su valor de retorno (mostrado en el eje "y" vertical de la gráfica) es, tal como se puede apreciar, el mismo que el que se obtendría al indicar un ángulo con un determinado valor dentro de ese rango. En cualquier caso, ya se ve que ese valor de retorno –que es de tipo *double*– es siempre uno entre -1 y 1; si se quiere ampliar ese rango, se deberá multiplicar entonces ese valor de retorno por una cantidad concreta (llamada "amplitud", A) para obtener así un valor entre -A y A.

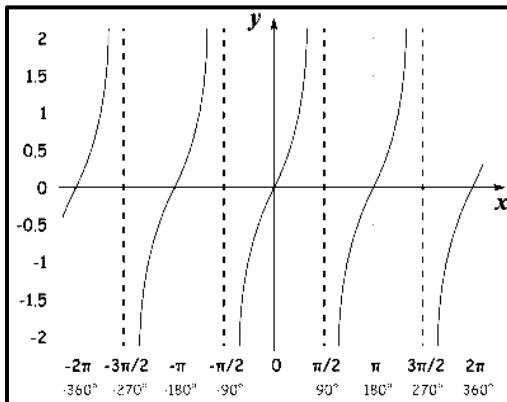


cos(): devuelve el coseno de un ángulo, especificado como único parámetro –de tipo *float*–. Este ángulo ha de venir expresado en radianes, por lo que sus valores posibles (mostrados en el eje "x" horizontal de la gráfica siguiente) a menudo estarán dentro del rango entre 0 y 2π ; si estuvieran fuera de él, al ser esta función periódica, su valor de retorno (mostrado en el eje "y" vertical de la gráfica) es, tal como se puede apreciar, el mismo que el que se obtendría al indicar un ángulo con un determinado valor dentro de ese rango. En cualquier caso, ya se ve que ese valor de retorno –que es de tipo *double*– es siempre uno entre -1 y 1; si se quiere ampliar ese rango, se deberá multiplicar entonces ese valor de retorno por una cantidad concreta (llamada "amplitud", A) para obtener así un valor entre -A y A.



tan(): devuelve la tangente de un ángulo, especificado como único parámetro –de tipo *float*–. Este ángulo ha de venir expresado en radianes, por lo que sus valores posibles (mostrados en el eje "x" horizontal de la gráfica siguiente) a

menudo estarán dentro del rango entre 0 y 2π ; si estuvieran fuera de él, al ser esta función periódica, su valor de retorno (mostrado en el eje "y" vertical de la gráfica) es, tal como se puede apreciar, el mismo que el que se obtendría al indicar un ángulo con un determinado valor dentro de ese rango. En cualquier caso, ya se ve que ese valor de retorno –que es de tipo *double*– es siempre uno entre $-\infty$ y ∞ .



También podemos utilizar números pseudoaleatorios en nuestros sketches Arduino. Un número pseudoaleatorio no es estrictamente un número aleatorio según la definición matemática rigurosa, pero para nuestros propósitos el nivel de aleatoriedad que alcanzan las siguientes funciones será más que suficiente:

randomSeed(): inicializa el generador de números pseudoaleatorios. Se suele ejecutar en la sección *void setup()* para poder utilizar a partir de entonces números pseudoaleatorios en nuestro sketch. Esta instrucción tiene un parámetro de tipo *long* llamado "semilla" que indica el valor a partir del cual empezará la secuencia de números. Semillas iguales generan secuencias iguales, así que interesaría en múltiples ejecuciones de *randomSeed()* utilizar valores diferentes de semilla para aumentar la aleatoriedad. También nos puede interesar a veces lo contrario: fijar la semilla para que la secuencia de números aleatorios se repita exactamente. No tiene ningún valor de retorno.

random(): una vez inicializado el generador de números pseudoaleatorios con *randomSeed()*, esta instrucción retorna un número pseudoaleatorio de tipo *long* comprendido entre un valor mínimo (especificado como primer parámetro –opcional–) y un valor máximo (especificado como segundo parámetro) menos uno. Si no se especifica el primer parámetro, el valor mínimo por defecto es 0. El tipo de ambos parámetros puede ser cualquiera mientras sea entero.

EL MUNDO GENUINO-ARDUINO

Un ejemplo de uso de las instrucciones trigonométricas sería el siguiente sketch. Si, una vez puesto en marcha, abrimos el "Serial Plotter", ahí podremos observar una figura similar a la gráfica de la función seno de la página anterior:

Ejemplo 4.18

```
float frecuencia = 2.0;
float amplitud = 100.0;
void setup() { Serial.begin(9600);}
void loop() {
    /*Calculo el valor del parámetro que tendrá la función sin(). Este valor será el tiempo transcurrido
    (calculado por millis() y reducido a segundos por la división entre 1000) a lo largo de la ejecución del
    sketch. La constante  $2\pi$  se ha añadido al cálculo para lograr que ese tiempo sea siempre un múltiplo de
    dicha cantidad, haciendo así más sencillo el control del periodo de la onda. El valor de la variable
    'frecuencia' sirve para definir el número de picos (o valles) de la onda que aparecen en un segundo.*/
    float x = 2 * PI * frecuencia * millis()/1000;
    //Envío (al "Serial Monitor"/"Serial Plotter" el valor devuelto por sin() tras ser ampliado
    Serial.println(amplitud * sin(x));
    //Espero 10ms para limitar el ritmo de actualización del Serial Plotter
    delay(10);
}
```

Un ejemplo de uso de las instrucciones pseudoaleatorias sería el siguiente sketch. Ahí se puede observar que se tiene una semilla fija y que se generarán números pseudoaleatorios entre 1 y 9. Precisamente por tener la semilla fija, si abrimos el "Serial monitor" en diferentes ocasiones (o el "Serial Plotter" si lo que queremos es obtener una representación gráfica), en todas ellas la secuencia de números obtenida será exactamente la misma (estamos suponiendo, ojo, que utilizamos un modelo de placa Arduino capaz de reiniciar el canal serie cada vez que abre el "Serial Monitor"/"Serial Plotter", tal como es el UNO):

Ejemplo 4.19

```
void setup(){
    Serial.begin(9600);
    randomSeed(100);
}
void loop(){
    Serial.println(random(1,10));
    delay(100);
}
```

Otro ejemplo (algo más útil) que emplea la generación de números pseudoaleatorios es el siguiente sketch, donde de una forma repetitiva (en un

intervalo de tiempo aleatorio entre medio segundo y tres segundos) se muestra por el canal serie diferentes mensajes elegidos al azar de entre un conjunto almacenado dentro de un array:

Ejemplo 4.20

```
char* mensajes[] = {"Un mensaje", "Otro", "Otro más", "Ya está bien"};
void setup(){ Serial.begin(9600); }
void loop(){
    int intervalo = random(500,3000);
    delay(intervalo);
    byte mensaje = random(4);
    Serial.println(mensajes[mensaje]);
}
```

Además de las instrucciones matemáticas anteriores, el lenguaje Arduino dispone de varios operadores aritméticos, algunos de los cuales ya han ido apareciendo en algunos códigos de ejemplo. Estos operadores funcionan tanto para números enteros como decimales y son los siguientes:

Operadores aritméticos

+	Operador suma
-	Operador resta
*	Operador multiplicación
/	Operador división
%	Operador módulo

El operador módulo sirve para obtener el resto de una división. Por ejemplo: $27 \% 5 = 2$. Es el único operador que no funciona con *floats*.

Un comentario final: posiblemente, a priori las funciones matemáticas del lenguaje Arduino pueden parecer escasas comparadas con las de otros lenguajes de programación. Pero nada más lejos de la realidad: precisamente porque el lenguaje Arduino no deja de ser un "maquillaje" del lenguaje C/C++, en realidad tenemos a nuestra disposición la mayoría de funciones matemáticas (y de hecho, prácticamente cualquier tipo de función) que ofrece el lenguaje C/C++. Concretamente, podemos utilizar todas las funciones listadas en la referencia online de "avr-libc" (<http://www.nongnu.org/avr-libc/user-manual>). Así pues, si necesitamos calcular el exponencial de un número decimal x , podemos usar $\exp(x)$; Si queremos calcular su

EL MUNDO GENUINO-ARDUINO

logaritmo natural, podemos usar `log(x)`; Si queremos calcular su logaritmo en base 10, podemos usar `log10(x)`; Si queremos calcular el módulo de una división de dos números decimales, podemos usar `fmod(x,y)`; etc.

INSTRUCCIONES DE GESTIÓN DE CADENAS

El lenguaje Arduino incluye un completo conjunto de instrucciones de manipulación y tratamiento de cadenas que, de una forma muy sencilla, permiten (por ejemplo) buscar una cadena dentro de otra, sustituir una cadena por otra, unir (o lo que es lo mismo, "concatenar") cadenas , eliminar parte de una cadena, conocer su longitud, etc., etc. No obstante, todas estas instrucciones solamente pueden ser utilizadas de la/s cadena/s a tratar que ha/n sido declarada/s previamente como *String*. Es decir: los arrays de caracteres no pueden ser manipulados por estas instrucciones.

Esto es así porque en realidad, cada una de las cadenas declaradas como *String* es tratada dentro del lenguaje Arduino como un objeto (al igual que el objeto *Serial*), en vez de como una variable "estándar", lo que permite, entre otras ventajas, integrarle instrucciones de manejo de forma inherente.

Una vez creada, pues, una cadena como objeto de clase *String* (supondremos que su valor inicial es "unacadena"), ya podremos utilizar sobre ella todas las instrucciones listadas a continuación:

unacadena.length(): devuelve la longitud de "unacadena" (es decir, el número de caracteres ASCII). No se cuenta el carácter nulo que marca los finales de cadena. No tiene parámetros.

unacadena.compareTo(): compara "unacadena" con otra que se pase como parámetro (bien de forma literal, bien mediante un objeto *String*). Comparar significa detectar qué cadena se ordena antes que la otra. Las cadenas se comparan desde su principio carácter a carácter utilizando el orden de sus códigos ASCII. Esto quiere decir, por ejemplo, que 'a' va antes que 'b' pero después que 'A', y las cifras van antes que las letras. Su valor de retorno será un número negativo si "unacadena" va antes que la cadena pasada como parámetro, 0 si las dos cadenas son iguales o un número positivo si "unacadena" va después que la cadena pasada como parámetro.

Si se quisiera realizar una comparación similar pero entre arrays de caracteres, se puede utilizar la instrucción **strcmp()**, la cual tiene dos parámetros (que corresponden con los dos arrays a comparar), y cuyo valor de retorno es idéntico a *unacadena.compareTo()*.

unacadena.equals(): compara si "unacadena" es igual a otra cadena, pasada como parámetro. La comparación es "case-sensitive": esto quiere decir que la cadena "hola" no es igual a "HOLA". Esta instrucción es equivalente al operador "==" para objetos *String* (este operador lo veremos dentro de poco). Su valor de retorno es *true* si "unacadena" es igual a la cadena especificada como parámetro, o *false* en caso contrario.

unacadena.equalsIgnoreCase(): compara si "unacadena" es igual a otra cadena, pasada como parámetro. La comparación es "case-insensitive": esto quiere decir que la cadena "hola" es igual a "HOLA". Su valor de retorno es *true* si "unacadena" es igual a la cadena especificada como parámetro, o *false* en caso contrario.

unacadena.indexOf(): devuelve la posición (un número entero) dentro de "unacadena" donde se encuentra el carácter o el principio de la cadena especificada como parámetro. Si no se encuentra nada, devuelve -1. Observar que las posiciones se numeran empezando por 0. Por defecto, la búsqueda comienza desde el principio de "unacadena", pero mediante un segundo parámetro opcional se puede indicar la posición a partir de la cual se quiere empezar a buscar. De esta manera, se puede utilizar esta instrucción para encontrar paso a paso todas las ocurrencias que existan de la cadena buscada.

unacadena.lastIndexOf(): devuelve la posición (un número entero) dentro de "unacadena" donde se encuentra el carácter o el principio de la cadena especificada como parámetro. Si no se encuentra nada, devuelve -1. Observar que las posiciones se numeran empezando por 0. Por defecto, la búsqueda comienza desde el final de "unacadena" hacia atrás, pero mediante un segundo parámetro opcional se puede indicar la posición a partir de la cual se quiere empezar a buscar (hacia atrás siempre). De esta manera, se puede utilizar esta instrucción para encontrar paso a paso todas las ocurrencias que existan de la cadena buscada.

unacadena.charAt(): devuelve el carácter cuya posición (dato entero) se haya especificado como parámetro. Las posiciones se numeran empezando por 0.

unacadena.setCharAt(): sustituye un determinado carácter (cuya posición dentro de "unacadena" se ha especificado como primer parámetro) por el carácter indicado como segundo parámetro. La posición del carácter a sustituir ha de ser un dato entero y su numeración empieza siempre por 0.

EL MUNDO GENUINO-ARDUINO

unacadena.substring(): devuelve la subcadena dentro de "unacadena" existente entre la posición inicial (especificada como primer parámetro) y la posición final (especificada como segundo parámetro opcional). La posición inicial indicada es inclusiva (es decir, el carácter que ocupa esa posición es incluido en la subcadena), pero la posición final –opcional– es exclusiva (es decir, el carácter que ocupa esa posición es el primero en no incluirse en la subcadena). Si dicha posición final no se especifica, la subcadena continúa hasta el final de "unacadena". Observar que las posiciones se numeran empezando por 0.

unacadena.replace(): sustituye una subcadena existente dentro de "unacadena" (especificada como primer parámetro) por otra (especificada como segundo), todas las veces que aparezca. También sirve para sustituir caracteres individuales. La sustitución se realiza sobre "unacadena", sobrescribiendo su valor original.

unacadena.remove(): elimina de "unacadena" una cantidad de caracteres indicada como segundo parámetro (en forma de número entero de tipo *word*), realizándose esta eliminación a partir del carácter ubicado en la posición indicada como primer parámetro (incluyéndolo). La numeración de esta posición empieza siempre desde 0. Se puede no especificar el segundo parámetro: en ese caso, la eliminación se realizaría hasta el final de "unacadena".

unacadena.toLowerCase(): convierte todos los caracteres de "unacadena" en minúsculas. La conversión se realiza sobre "unacadena", sobrescribiendo su valor original.

unacadena.toUpperCase(): convierte todos los caracteres de "unacadena" en mayúsculas. La conversión se realiza sobre "unacadena", sobrescribiendo su valor original.

unacadena.trim(): elimina todos los espacios en blanco y tabulaciones existentes al principio y al final de "unacadena"; más en concreto, elimina los caracteres ASCII siguientes: nº 32 (espacio en blanco), nº 9 (tabulador), nº 10 (nueva línea), nº 11 (tabulador vertical), nº 12 (salto de página) y nº 13 (retorno de carro). La conversión se realiza sobre "unacadena", sobrescribiendo su valor original.

unacadena.concat() : añade –"concatena"– al final de la cadena "unacadena" otra cadena, pasada como parámetro. Como resultado obtendremos un nuevo valor en "unacadena": su valor original seguido de ese valor pasado por parámetros, unidos. Es equivalente al operador "+" para objetos String.

Si se quisiera realizar una concatenación similar pero entre arrays de caracteres, se puede utilizar la instrucción **strcat()**, la cual tiene dos parámetros: el primero es un array de caracteres cuyo significado es equivalente al valor "unacadena" descrito en el párrafo anterior y el segundo es otro array de caracteres equivalente al parámetro de *unacadena.concat()*. Otra instrucción interesante, por otro lado, es **strcpy()**, la cual, en vez de concatenar, sobrescribe el valor actual del array de caracteres indicado como primer parámetro por la cadena indicada como segundo.

unacadena.endsWith(): chequea si "unacadena" acaba con los caracteres de otra cadena, pasada por parámetro. Su valor de retorno es *true* si "unacadena" acaba con la cadena especificada como parámetro, o *false* en caso contrario.

unacadena.startsWith() : Chequea si "unacadena" empieza con los caracteres de otra cadena, pasada por parámetro. Su valor de retorno es *true* si "unacadena" empieza con la cadena especificada como parámetro, o *false* en caso contrario.

unacadena.toCharArray(): copia una cantidad determinada de caracteres pertenecientes a "unacadena" a un array de tipo *char*. Ese array ha de ser especificado como primer parámetro, y la cantidad de caracteres a copiar allí ha de ser especificada como segundo parámetro –de tipo *word*– . Siempre se empiezan a obtener los caracteres desde el principio de "unacadena". No tiene valor de retorno.

unacadena.getBytes(): copia una cantidad determinada de caracteres pertenecientes a "unacadena" a un array de tipo *byte*. Ese array ha de ser especificado como primer parámetro, y la cantidad de caracteres a copiar allí ha de ser especificada como segundo parámetro –de tipo *word*– . Siempre se empiezan a obtener los caracteres desde el principio de "unacadena". No tiene valor de retorno.

unacadena.toInt(): si "unacadena" tiene un valor que empieza por cifras numéricas, esta instrucción es capaz de distinguirlas (descartando los posibles caracteres no numéricos posteriores) y devolver ese valor numérico transformado en un dato de tipo entero. Es decir, transforma una cadena en un número entero, si es posible. Esta instrucción no tiene parámetros.

unacadena.toFloat(): si "unacadena" tiene un valor que empieza por cifras numéricas y contiene entre ellas el símbolo del punto decimal ('.'), esta instrucción es capaz de distinguir tanto las cifras como el punto (hasta que

EL MUNDO GENUINO-ARDUINO

encuentre el primer carácter no numérico) y devolver ese valor numérico transformado en un dato de tipo *float*. Es decir, transforma una cadena en un número decimal. Ese número decimal obtenido tendrá siempre dos cifras decimales que se redondearán si fuera necesario. Por ejemplo, la cadena "123.456holo" es convertida en el número 123.46. Si el primer carácter de "unacadena" ya no fuera una cifra numérica, se devolverá 0. Esta instrucción no tiene parámetros.

A continuación, se muestra un sketch de ejemplo donde se puede observar el comportamiento de la mayoría de instrucciones descritas anteriormente:

Ejemplo 4.21

```
String unacadena="En un lugar de La Mancha de cuyo nombre";
String otracadena="Erase una vez";
byte pos=0;
String cadenaEntero="13asdf";
void setup(){
    Serial.begin(9600);
    Serial.println(unacadena.length());           //Devolverá 39
    Serial.println(unacadena.endsWith("nombre")); //Devolverá 1 (true)
    Serial.println(unacadena.startsWith("En"));   //Devolverá 1 (true)
    //Devolverá un nº negativo ('n' va antes de 'r')
    Serial.println(unacadena.compareTo(otracadena));
    Serial.println(unacadena.equals(otracadena)); //Devolverá 0 (son diferentes)
    Serial.println(unacadena.equalsIgnoreCase(otracadena)); //Devolverá 0 (son diferentes)
    //Primero devolverá 12 (primer "de") y luego 25 (segundo "de")
    pos=unacadena.indexOf("de");
    Serial.println(pos);
    Serial.println(unacadena.indexOf("de",pos+1));
    //Devolverá -1 porque se empieza a buscar antes
    Serial.println(unacadena.lastIndexOf("Mancha",3));
    Serial.println(unacadena.charAt(1));           //Devolverá 'n'
    Serial.println(unacadena.substring(3,6));       //Devolverá "un"
    Serial.println(unacadena.substring(unacadena.indexOf("cuyo"))); //Devolverá "cuyo nombre"
    unacadena.replace("nombre","apellido");        //La palabra "apellido" sustituye a "nombre"
    Serial.println(unacadena);
    unacadena.remove(11);                         //La cadena pasa a ser "En un lugar"
    Serial.println(unacadena);
    unacadena.setCharAt(0,'T');                   //La letra 'T' sustituye a la primera letra ('E')
    Serial.println(unacadena);
    unacadena.toUpperCase();                     //Toda la cadena se transforma en mayúsculas
    Serial.println(unacadena);
    unacadena.trim();                           //Se eliminan los espacios en blanco en los extremos
```

```

Serial.println(unacadena);
unacadena.concat(otracadena);           // "otracadena" se concatena con "unacadena"
Serial.println(unacadena);
//Devolverá "13asdf7" ("+" actúa como concatenador de cadenas)
Serial.println(cadenaEntero+7);
/*La instrucción toInt() convierte la cadena "13asdf7" en el número entero 13,
por lo que "+" ahora actúa como operador suma */
Serial.println(cadenaEntero.toInt() +7);
}
void loop(){}

```

CREACIÓN DE INSTRUCCIONES (FUNCIONES) PROPIAS

Imaginemos que tenemos un conjunto de instrucciones que hemos de escribir repetidas veces en diferentes partes de nuestro sketch. ¿No habría alguna manera para poder invocar a ese conjunto de instrucciones mediante un simple nombre sin tener que volver a escribir todas ellas cada vez? Sí, mediante la creación de funciones. Una función es un trozo de código al que se le identifica con un nombre. De esta forma, se puede ejecutar todo el código incluido dentro de ella simplemente escribiendo su nombre en el lugar deseado de nuestro sketch.

Al crear nuestras propias funciones, escribimos código mucho más legible y fácil de mantener. Segmentar el código en diferentes funciones permite al programador crear piezas modulares de código que realizan una tarea definida. Además, una función la podemos reutilizar en otro sketch, de manera que con el tiempo podemos tener una colección muy completa de funciones que nos permitan escribir código muy rápida y eficientemente.

Resumiendo, incluir fragmentos de código en funciones tiene diversas ventajas: las funciones ayudan al programador a ser organizado (a menudo esto ayuda a conceptualizar el programa), codifican una acción en un lugar, de manera que una función solo ha de ser pensada y escrita una vez (esto también reduce las posibilidades de errores en una modificación, si el código ha de ser cambiado) y hacen más fácil la reutilización de código en otros programas (provocando que estos sean más pequeños, modulares y legibles).

Para crear una función propia, debemos declararlas. Esto se hace en cualquier lugar fuera de *void setup()* y *void loop()* –por tanto, bien antes o después de ambas secciones–, siguiendo la sintaxis marcada por la plantilla siguiente:

EL MUNDO GENUINO-ARDUINO

```
tipoRetorno nombreFuncion (tipo param1, tipo param2,...) {  
    // Código interno de la función, el cual...  
    // ...puede incluir opcionalmente la instrucción especial return valor;  
}
```

donde:

"**tipoRetorno**" es uno de los tipos ya conocidos (*byte, int, float*, etc.) e indica el tipo de valor que la función devolverá al sketch principal una vez ejecutada. Este valor devuelto se podrá guardar en una variable para ser usada en el sketch principal, o simplemente puede ser ignorado. Si no se desea devolver ningún dato (es decir: que la función realice su tarea y punto), se puede utilizar como tipo de retorno uno especial llamado *void* o bien no especificar ninguno. Para devolver el dato, se ha de utilizar la instrucción *return valor;*, la cual tiene como efecto "colateral" el fin de la ejecución de la función. Esto hace que normalmente esta instrucción sea la última en escribirse dentro del código de la función. Si la función no retorna nada (es decir, si el tipo de retorno es *void*), no es necesario escribirla.

"**tipo param1, tipo param2,...**" son las declaraciones de los parámetros de la función, que no son más que variables internas cuya existencia solo perdura mientras el código de esta se esté ejecutando. Es decir, variables locales de esa función. El valor inicial de estos parámetros se asigna explícitamente en la "llamada" a la función (esto es, cuando esta es invocada dentro del sketch principal), pero este valor puede variar dentro de su código interno. En todo caso, al finalizar la ejecución de la función, todos sus parámetros (y de hecho, cualquier otra variable local que se haya declarado dentro de ella) son destruidos de la memoria del microcontrolador. El número de parámetros puede ser cualquiera: ninguno, uno, dos, etc.

NOTA: Debido a la propia naturaleza de las variables locales, hay que tener presente que en cada nueva llamada de la función a la que pertenecen volverán a inicializarse de nuevo como si fuera la primera vez.

Fijarse, además, que el código interno de la función está delimitado por las llaves de apertura y cierre.

Una vez hayamos declarado las funciones deseadas (ya sea antes o después de las secciones *void setup()* y *void loop()*, eso da igual), ya podremos invocarlas (es decir, ejecutar su código interno) en cualquier parte de nuestro sketch simplemente escribiendo *nombreFuncion(valorInicialParam1,valorInicialParam2,...);* o bien, si

deseáramos guardar su valor de retorno en una variable –ya declarada–, escribiendo `mivariable= nombreFuncion(valorInicialParam1,valorInicialParam2,...);`. En cualquier caso, tras invocar a una función (y ejecutarse, por tanto, su código interno), el sketch continúa su marcha justo en la línea posterior a donde está escrita dicha invocación.

El valor inicial de un parámetro (es decir, el asignado directamente en la llamada de la función –lo que vendría a representar `valorInicialParam1` en el párrafo anterior–) puede ser indicado explícitamente o bien a través del nombre de una variable –de su mismo tipo! – que contenga en el momento de la llamada el valor en cuestión. En este segundo caso, es importante tener en cuenta que el valor del parámetro y el valor de la variable utilizada son completamente independientes, de forma que si el valor del parámetro fuera modificado dentro del código interno de la función esto no afectaría para nada al valor almacenado en la variable. Esto es así porque en realidad el valor del parámetro es una copia del de la variable y la función siempre trabaja sobre esa copia. Este comportamiento en la literatura técnica se denomina "paso por valor" y el siguiente código de ejemplo permite observarlo:

Ejemplo 4.22

```
byte a = 3;      //Variable global (cuyo valor no va a ser modificado en ningún momento)
void mostrarNumeros(byte x, byte y){      //Esta función no tiene valor de retorno
    //Muestro el valor pasado en la llamada de la función a 'x' (valor directo = 6)
    Serial.println(x);
    //Muestro el valor pasado en la llamada de la función a 'y' (valor de la variable global 'a' = 3)
    Serial.println(y);
    //Modifico el valor del parámetro 'y', pero esto no afecta para nada a la variable global 'a'
    y = y + 1;
    Serial.println(y); //Muestro 4
    Serial.println(a); //Sigo mostrando 3
}
void setup(){
    Serial.begin(9600);
    /*Paso el valor inicial de los dos parámetros en la llamada
    (uno directamente y otro mediante la variable 'a') */
    mostrarNumeros(6,a);
}
void loop(){}
```

Veamos ahora un ejemplo de función con valor de retorno. Concretamente, en el siguiente ejemplo se crea una función que devuelve el resultado de multiplicar dos números pasados como parámetros:

EL MUNDO GENUINO-ARDUINO

Ejemplo 4.23

```
int multiplicar(int x, int y){  
    /*La variable 'resultado', al estar declarada dentro de la función 'multiplicar()',  
    no existe fuera de ella. Es la encargada de guardar el valor de retorno, devuelto gracias  
    a la instrucción 'return' */  
    int resultado;  
    /*'x' e 'y' valdrán lo que se ponga en cada invocación que se realice de la función  
    'multiplicar()' en el sketch principal (en este caso, 'x'='primernumero' e  
    'y'='segundonumero'), es decir 'x'=2 y 'y'=3*/  
    resultado = x * y ;  
    //¡¡El tipo de la variable 'resultado' ha de coincidir con el tipo de retorno ('int') !!  
    return resultado;  
}  
void setup(){ Serial.begin(9600); }  
void loop(){  
    int primernumero=2;  
    int segundonumero=3;  
    int recojoresultado;  
    /*Los valores pasados como parámetros han de ser del mismo tipo que los especificados en la  
    declaración de la función. Por otro lado, a la variable 'recojoresultado', propia del sketch principal, se le  
    asignará el valor que retorna la función 'multiplicar()' gracias a su variable interna 'resultado';  
    lógicamente,'recojoresultado' y 'resultado' han de ser del mismo tipo también */  
    recojoresultado=multiplicar(primernumero,segundonumero);  
    Serial.println(recojoresultado);  
}
```

Uno podría pensar que el uso de parámetros en una función no es necesario porque mediante variables globales cualquier función podría acceder a los valores necesarios para trabajar. Es decir, podríamos haber modificado el código 4.22 dejándolo así:

Ejemplo 4.24

```
int primernumero=2;  
int segundonumero=3;  
int multiplicar(){  
    int resultado;  
    resultado = primernumero * segundonumero ;  
    return resultado;  
}  
void setup(){  
    Serial.begin(9600);  
}  
void loop(){
```

```

int recojoresultado;
recojoresultado=multiplicar();
Serial.println(recojoresultado);
}

```

En el código anterior, tanto *primernumero* como *segundonumero* se han declarado como variables globales y la función *multiplicar()* no ha necesitado por tanto ningún parámetro. No obstante, el problema de usar variables globales en vez de pasar parámetros a la función es la pérdida de elegancia y flexibilidad. ¿Qué pasaría si además de querer multiplicar 2x3 quisiéramos multiplicar 4x7? En el caso de utilizar parámetros, sería tan sencillo como pasar los nuevos valores directamente como parámetros (así: *multiplicar(4,7);*). En cambio, al utilizar variables globales necesitaríamos cambiar sus valores cada vez que quisiéramos multiplicar diferentes números (o tener varias variables globales para cada número que deseáramos multiplicar, algo bastante inviable), con lo que podríamos alterar el comportamiento de otras partes de nuestro código y ser una fuente de errores.

Finalmente, indicar que aunque hasta ahora hemos ido llamando "instrucción" a los diferentes comandos del lenguaje Arduino (algunos dentro de objetos y otros no), en realidad, todas estas instrucciones son también funciones. Funciones que permiten invocar un conjunto de código escrito en lenguaje C invisible para nosotros. Incluso lo que hemos venido llamando la "sección" *void setup()* y la "sección" *void loop()* también son funciones (sin parámetros y sin valor de retorno, como se puede ver).

Funciones con parámetrosopcionales ("sobrecarga")

El lenguaje Arduino no permite definir funciones con parámetros opcionales. Es decir, si en la declaración de una determinada función especificamos (por ejemplo) que esta tiene tres parámetros, cada vez que la invoquemos deberemos indicar obligatoriamente tres valores, correspondientes a cada uno de esos tres parámetros, ni uno más ni uno menos. A veces, no obstante, nos puede interesar invocar a esa misma función indicando menos valores (dos, uno o ninguno, en este ejemplo), de forma que los no indicados tengan un valor por defecto concreto; esto nos permitiría tener nuestro código más claro y más versátil. Afortunadamente, podemos utilizar un truco ingenioso para conseguir esto que consiste en declarar una función (con el número de parámetros máximo que necesitemos) que realice la acción deseada y, además, declarar otra función con el mismo nombre pero con parámetros omitidos y que en su interior invoque a la primera indicándole en ese momento los valores concretos deseados para esos parámetros omitidos.

EL MUNDO GENUINO-ARDUINO

En el siguiente código se muestra un ejemplo de lo descrito: en él se declara una función llamada *mensaje()* que tiene dos parámetros y otra función homónima que solo tiene uno (correspondiente al segundo parámetro de la primera). En realidad, lo que hace esta segunda función simplemente es llamar a la primera función indicando siempre como valor de su primer parámetro un determinado valor fijo. De esta forma, conseguimos poder invocar a "la misma función" *mensaje()* de dos formas distintas (con uno o dos parámetros). En la literatura técnica a esta posibilidad se le suele llamar "sobrecarga de funciones":

Ejemplo 4.25

```
void mensaje(String saludo, String despedida){  
    Serial.println(saludo);  
    Serial.println(despedida);  
    Serial.println("-----");  
}  
void mensaje(String despedida){  
    mensaje("Buenas tardes", despedida);  
}  
void setup() {  
    Serial.begin(9600);  
}  
void loop() {  
    mensaje("Hola","Hasta luego");  
    delay(500);  
    mensaje("Hasta pronto"); //El mensaje de saludo siempre es "Buenas tardes"  
    delay(500);  
}
```

Funciones con estructuras como parámetros o valor de retorno

En el caso de querer declarar una función que tenga algún parámetro de tipo estructura (o que su valor de retorno sea de ese tipo), debemos tener en cuenta que no podremos escribir la declaración de esa función dentro del mismo sketch donde se vaya a invocar sino en un fichero separado. Esto es así porque en el proceso interno de compilación de un sketch primero se reconocen todas las declaraciones de funciones existentes, estén escritas donde estén escritas (al principio, medio o final del sketch) y a continuación se reconocen los nuevos tipos de datos personalizados (es decir, las estructuras); debido a que el compilador sigue este orden, si encontrara una declaración de función donde hubiera implicada una estructura (como parámetro o como valor de retorno), no sabría lo que es (porque todavía no ha llegado a reconocerla) y lanzaría un error. El truco para obligar al compilador a reconocer las declaraciones de funciones después de las declaraciones de las estructuras es escribir,

tal como se ha dicho, las primeras en un fichero aparte (llámémoslo "funciones.h"), donde la extensión ".h" es un convenio estándar para este tipo de ficheros) y, además, escribir la instrucción especial `#include "funciones.h"` dentro de nuestro sketch justo después de las creaciones de nuevos tipos de datos (supondremos que "funciones.h" se encuentra dentro de la misma carpeta donde se halla el fichero ".ino" de nuestro sketch). Utilizando la instrucción `#include` estamos obligando al compilador a reconocer el contenido del fichero indicado (en este caso, las declaraciones de las funciones) justo en el lugar exacto donde la hayamos escrito: así pues, si la escribimos después de las definiciones de estructuras, habremos conseguido alterar el orden de "reconocimiento" a nuestra conveniencia.

Pongamos un ejemplo: el siguiente código muestra la declaración de una función que recibe como parámetro una estructura (es decir, un tipo de datos personalizado) llamada *sensor* formada por dos elementos llamados *nombre* y *valor* (se supone que correspondientes al nombre de un sensor y al valor obtenido por este de una determinada medida). Por ello deberá ser escrita en un fichero separado de nuestro sketch principal (que llamaremos "mifunc.h" y guardaremos en la misma carpeta donde se halle aquel... ambas acciones se pueden conseguir seleccionando la opción de "New tab" –"Nueva pestaña"– del IDE Arduino):

Ejemplo 4.26

```
void mensaje(sensor s){
    Serial.println(s.nombre);
    Serial.println(s.valor);
    Serial.println("----");
}
```

Y a continuación mostramos el código del sketch que invoca a la función *mensaje()* anterior. Nótese la ubicación de la línea `#include` justo después de la creación del nuevo tipo de datos *sensor*:

Ejemplo 4.26 (BIS)

```
struct sensor { String nombre; int valor; };
#include "a.h"
void setup() {
    Serial.begin(9600);
}
void loop() {
    //El valor del parámetro de 'mensaje' puede ser un dato-estructura pasado directamente...
    mensaje({ "Sensor1", 376 });
    delay(500);
}
```

```
//...o bien un dato previamente declarado como estructura  
sensor z={"Sensor2",9573};  
mensaje(z);  
delay(500);  
}
```

No está de más comentar que, al igual que hemos utilizado los ficheros ".h" para incluir en nuestro código funciones que manejan estructuras (y así solventar un problema concreto relacionado con su procesamiento), nada nos impide emplear estos mismos ficheros ".h" como almácén de declaraciones de cualquier otro tipo de funciones sin que tengan que tener estas ningún "problema" en particular. Esto es, de hecho, lo que básicamente son las librerías.

Funciones con más de un valor de retorno ("paso por referencia")

Todos los ejemplos que hemos visto hasta ahora muestran funciones propias cuyos parámetros reciben su valor inicial en el momento de la "llamada". Una vez inicializados, estos parámetros se comportan dentro del código interno de la función como variables locales estándar hasta que dicho código interno acaba su ejecución: en ese momento, los parámetros desaparecen de la memoria del microcontrolador y el valor que tenían en ese momento se pierde. Como ya sabemos, las variables definidas fuera de una función no pierden nunca su valor cuando esta finaliza su ejecución, así que uno podría preguntarse si sería posible conseguir algo parecido con el valor de un parámetro: no perderlo tras el fin de la ejecución de la función que lo define. La solución más obvia sería utilizar variables globales para almacenar los valores de los parámetros deseados, pero hay una técnica más elegante y óptima que requiere solo tres pasos:

1. Indicar en la declaración de la función en cuestión (llamémosla *blabla()*) cuáles de sus parámetros queremos que se comporten de esta manera "especial" (es decir, qué valores queremos que no se pierdan una vez finalizada la ejecución de su código interno). Esta indicación se realiza simplemente precediendo el nombre de los parámetros deseados con el signo "&". Estos parámetros "marcados" son los que técnicamente se llaman "pasados por referencia", en contraposición a los parámetros "comunes", que se denominan "pasados por valor").
2. Declarar, por cada parámetro pasado por referencia, una variable local de su mismo tipo dentro de la sección de nuestro sketch desde donde *blabla()* será invocada (generalmente, esa sección será *void loop()* pero puede ser cualquier otra). El propósito de estas variables es

precisamente el de guardar los valores finales de esos parámetros tras la ejecución de *blabla()*.

3. Asociar sin ambigüedades cada variable declarada en el punto anterior a un parámetro concreto de *blabla()*. Esto se realiza en el mismo momento de la invocación de *blabla()* indicando como valor de cada parámetro pasado por referencia el nombre de su variable asociada.

El siguiente código muestra un ejemplo de lo recién descrito. En él utilizamos una variable (*x*) asociada a un parámetro pasado por referencia (*dato1*) de la función *suma()*, la cual tiene además otro parámetro pasado por valor (*dato2*). Lo que conseguimos con el paso por referencia es guardar de forma permanente en *x* los diferentes cambios que sufra internamente *dato1* (concretamente, la suma del valor de *dato2*). Por eso, antes de la ejecución de *suma()*, vemos en el "Serial monitor" un determinado valor de *x* y después, otro. De hecho, este comportamiento puede entenderse como si *suma()* tuviera un valor de retorno (a pesar de estar declarada como *void*) ya que, en la práctica, en cada ejecución de *suma()* estamos devolviendo un valor modificado *x* listo para ser usado en nuestro sketch. Esta idea podría extenderse al empleo de múltiples parámetros pasados por referencia, consiguiendo así que una función pudiera tener más de un valor de retorno:

Ejemplo 4.27

```
void setup() {
    Serial.begin(9600);
}
void loop(){
    byte x = random(10);
    Serial.print("El valor inicial de 'x' es: ");
    Serial.println(x);
    suma(x,3);
    Serial.print("El valor final de 'x' es: ");
    Serial.println(x);
    delay(2000);
}
void suma(byte &dato1, byte dato2) {
    dato1 = dato1 + dato2;
}
```

Es muy interesante ser consciente de que si eliminamos del código anterior el símbolo **&**, convertiremos "dato1" en un parámetro pasado por valor. Esto quiere decir que *dato1* será desintegrado al finalizar la ejecución del código interno de

EL MUNDO GENUINO-ARDUINO

suma() y, por tanto, su valor desaparecerá. Esto hará que *x* se comporte "de la manera habitual" y siga teniendo su valor inicial inalterado (tal como se podría observar de nuevo en el "Serial monitor") al no verse afectado por ningún cambio dentro del código interno de *suma()*.

Las variables *static*

Hasta ahora solamente hemos distinguido entre dos tipos de variables: las variables globales (aquellas que son manipulables —y por tanto, sobreescrivibles— desde cualquier línea de código de nuestro sketch, ya sea dentro de las funciones *setup()*, *loop()* o dentro de alguna función propia creada por nosotros) y las variables locales (aquellas que únicamente son manipulables dentro del mismo bloque de código donde fueron declaradas, ya sea este una función propia como un bloque condicional o bucle —estudiados en el siguiente apartado—). No obstante, existe un tercer tipo de variables que nos pueden ser muy útiles en determinadas circunstancias: las variables *static* ("estáticas").

Una variable estática es una variable local. Por tanto, únicamente puede ser manipulada dentro de la función donde se ha declarado. Pero a diferencia de las variables locales propiamente dichas, las cuales son destruidas (y por tanto, "desaparecen sin dejar rastro") cuando finaliza la ejecución de la función a la que pertenecen (volviéndose a inicializar siempre de cero cada vez que se vuelve a llamar a esa función otra vez), las variables *static* mantienen el valor que tenían al finalizar la ejecución de la función a la que pertenecen, de manera que a la siguiente llamada de esa función, empiezan teniendo el valor conseguido en la vez anterior. En este sentido, se comportarán más como una variable global (al persistir entre diferentes llamadas a una función y, por tanto, no perder su valor en ningún momento), pero con una gran diferencia: solamente son manipulables dentro de la función donde son declaradas y no desde el resto del código.

La decisión de utilizar variables *static* o variables globales dependerá sobre todo de si la variable en cuestión necesita estar accesible desde varias funciones diferentes de nuestro código. Si es que sí, no habrá más remedio que usar variables globales. Pero si es que no, lo recomendable es utilizar variables *static* porque así se evitan posibles modificaciones accidentales realizadas por error desde otros puntos del código (en proyectos de cierta magnitud, estas circunstancias son relativamente habituales).

A continuación mostramos un código de ejemplo donde se ilustra el comportamiento de este tipo de variables:

Ejemplo 4.28

```

void setup(){
    Serial.begin(9600);
}
void loop(){
    unaFuncion();
    otraFuncion();
    delay(200);
    /*Serial.println(unavar); ERROR de compilación: 'unavar' was not declared in this scope */
}
/*La primera vez que se invoca a unaFuncion(), se realiza la declaración e inicialización de 'unavar', asignándole el valor 3. Seguidamente, se le aumenta en una unidad su valor y se imprime por el "Serial monitor" (por tanto, inicialmente se verá el valor 4). En la siguiente iteración del loop() se vuelve a invocar a unaFuncion(), pero como 'unavar' es static, la inicialización ya no se produce sino que se continúa la ejecución de unaFuncion() con el valor que quedó almacenado en la anterior invocación -esto es, 4-. Por tanto, la línea unavar=unavar +1 asignará a la variable un nuevo valor igual a 5 (y lo imprimirá en el "Serial monitor"). Y en la siguiente invocación, volverá a pasar lo mismo y 'unavar' llegará a valer 6. Y así, hasta que se llegue al límite del tipo byte -255- y se vuelva a empezar desde 0. En resumen: la inicialización de una variable static tan solo se produce la primera vez que se llama a la función donde está definida.*/
void unaFuncion(){
    static byte unavar=3;
    unavar = unavar + 1;
    Serial.println(unavar);
}
/*Como la variable 'unavar' de otraFuncion() no es static, será reinicializada cada vez que se llame a dicha función. Por tanto, siempre se mostrará por el "Serial monitor" el valor 4. Notar que las dos variables (la definida en unaFuncion() y la definida en otraFuncion()) tienen el mismo nombre 'unavar' y no hay ningún problema, ya que ambas solamente son accesibles dentro del código de su propia función (una por ser static y otra por ser local) y, por tanto, no hay solapamiento. */
void otraFuncion(){
    byte unavar=3;
    unavar = unavar + 1;
    Serial.println(unavar);
}

```

BLOQUES CONDICIONALES**Los bloques *if* e *if/else***

Un bloque *if* sirve para comprobar si una condición determinada es cierta (es decir: *true* o 1) o falsa (es decir: *false* o 0). Si la condición es cierta, se ejecutarán las instrucciones escritas en su interior (es decir, dentro de las llaves de apertura y

EL MUNDO GENUINO-ARDUINO

cierre). Si no se cumple, puede no pasar nada, o bien, si existe tras el bloque *if* un bloque *else* (opcional), se ejecutarán las instrucciones escritas en el interior de ese bloque *else*. Es decir, si solo escribimos el bloque *if*, el sketch tendrá respuesta solamente para cuando sí se cumple la condición; pero si además escribimos un bloque *else*, el sketch tendrá respuesta para cuando sí se cumple la condición y para cuando no se cumple también. En general, la sintaxis del bloque *if/else* es:

```
if (condición) {  
    //Instrucciones –una o más– que se ejecutan si la condición es cierta  
} else {  
    //Instrucciones –una o más– que se ejecutan si la condición es falsa  
}
```

Tanto si se ejecutan las sentencias del bloque *if* como si se ejecutan las sentencias del bloque *else*, cuando se llega a la última línea de esa sección (una u otra), se salta a la línea inmediatamente posterior a su llave de cierre para continuar desde allí la ejecución del programa.

Hemos dicho que el bloque *else* es opcional. Si no lo escribimos, el *if* quedaría así:

```
if (condición) {  
    //Instrucciones –una o más– que se ejecutan si la condición es cierta  
}
```

En este caso, si la condición fuera falsa, el interior del *if* no se ejecutaría y directamente se pasaría a ejecutar la línea inmediatamente posterior a su llave de cierre (por lo que, tal como ya hemos comentado, el programa no hace nada en particular cuando la condición es falsa).

También existe la posibilidad de incluir una o varias secciones *else if*, siendo en este caso también opcional el bloque *else final*. Esta construcción tendría la siguiente sintaxis (puede haber todos los *else if* que se deseen):

```
if (condición) {  
    //Instrucciones –una o más– que se ejecutan si la condición es cierta  
} else if (otra_condición) {  
    /*Instrucciones –una o más– que se ejecutan si la condición  
    del anterior "if" es falsa pero la actual es cierta */  
} else if (otra_condición) {  
    /*Instrucciones –una o más– que se ejecutan si la condición
```

```

    del anterior "if" es falsa pero la actual es cierta */
} else {
    //Instrucción(es) que se ejecutan si todas las condiciones anteriores eran falsas
}

```

Es posible anidar bloques *if* uno dentro de otro sin ningún límite (es decir, se pueden poner más bloques *if* dentro de otro bloque *if* o *else*, si así lo necesitamos).

Ahora que ya sabemos las diferentes sintaxis del bloque *if*, veamos qué tipo de condiciones podemos definir entre los paréntesis del *if*. Lo primero que debemos saber es que para escribir correctamente en nuestro sketch estas condiciones necesitaremos utilizar alguno de los llamados operadores de comparación, que son los siguientes.

Operadores de comparación

<code>==</code>	Comparación de igualdad
<code>!=</code>	Comparación de diferencia
<code>></code>	Comparación de mayor que
<code>>=</code>	Comparación de mayor o igual que
<code><</code>	Comparación de menor que
<code><=</code>	Comparación de menor o igual que

Sabiendo esto, ya podemos escribir todas las condiciones que deseemos. A continuación, se muestra un código de ejemplo:

Ejemplo 4.29

```

int numero;
void setup(){
    Serial.begin(9600);
}
void loop(){
    if (Serial.available() > 0){
        //El nº introducido ha de estar en el rango del tipo 'int'
        numero=Serial.parseInt();
        if (numero == 23){
            Serial.println("Número es igual a 23");
        } else if (numero < 23) {
            Serial.println("Número es menor que 23");
        }
    }
}

```

```
    } else {
        Serial.println("Numero es mayor que 23");
    }
}
```

En el código anterior aparece un primer *if* que comprueba si hay datos en el buffer de entrada del chip TTL-UART pendientes de leer. Si es así (es decir, si el valor devuelto por *Serial.available()* es mayor que 0), se ejecutarán todas las instrucciones en su interior. En cambio, si la condición resulta ser falsa (es decir, si *Serial.available()* devuelve 0 y por tanto no hay datos que leer), fíjarse que la función *loop()* no ejecuta nada. En el momento que la condición sea verdadera, lo que tenemos dentro del primer *if* es la función *Serial.parseInt()* que reconoce y extrae de todo lo que se haya enviado a través del canal serie (por ejemplo, usando el "Serial monitor") un número entero, asignándolo a la variable *numero*. Y aquí es cuando llega otro *if* que comprueba si el valor de *numero* es igual a 23. Si no es así, se comprueba entonces si su valor es menor de 23. Y si todavía no es así, solo queda una opción: que sea mayor de 23. Al ejecutar este sketch a través del "Serial monitor", lo que veremos es que cada vez que envíemos algún dato a la placa, esta nos responderá con alguna de las tres posibilidades.

En general, el *if* es capaz de comparar números decimales también: en el ejemplo anterior, si *número* fuera una variable *float*, sustituyendo *Serial.parseInt()* por *Serial.parseFloat()* obtendríamos el mismo comportamiento. En el caso de las cadenas, no obstante, hay que tener en cuenta que solamente se pueden utilizar los operadores de comparación cuando ambas cadenas han sido declaradas como objetos *String*: si son arrays de caracteres no se pueden comparar. En este sentido, comparar si una cadena es mayor o menor que otra simplemente significa evaluar las cadenas en orden alfabético carácter tras carácter (por ejemplo, "alma" sería menor que "barco", pero "999" es mayor que "1000" ya que el carácter '9' va después del '1'). Recalcar que las comparaciones de cadenas son case-sensitive (es decir, el dato de tipo *String* "hola" no es igual que el dato de tipo *String* "HOLA").

Como demostración de lo explicado en el párrafo anterior, en el siguiente sketch se ilustra el uso del operador de igualdad, el cual en este caso resulta funcionalmente equivalente a escribir la expresión `cad1.equals(cad2)`:

Ejemplo 4.30

```
String cad1="hola";  
String cad2="hola";
```

```

void setup(){
    Serial.begin(9600);
}
void loop(){
    if(cad1 == cad2){
        Serial.println("La cadena es igual");
    } else {
        Serial.println("La cadena es diferente");
    }
}

```

Por otro lado, es bastante probable encontrar en bastantes códigos expresiones tales como *if(mivariable)* en vez de *if(mivariable!=0)*, por ejemplo. Es decir, *ifs* que solo incluyen una variable, pero no la condición para evaluarla. Esta forma de escribir los *ifs* es simplemente un atajo: si en el *if* solo aparece una variable o expresión sin ningún tipo de comparación, es equivalente a comprobar si dicha variable o expresión es *true* (es decir, si es diferente de 0). Si se diera el caso de que en el *if* aparece tan solo una función, sería equivalente a comprobar si el valor que devuelve esa función es, también, diferente de 0.

Seguramente sorprenderá que el operador de igualdad ("==") sea un doble igual. Esto es debido a que el signo igual individual ("=") representa el operador de asignación. Por lo tanto, ambos símbolos tienen un significado totalmente diferente, y pueden dar muchos problemas para quien se despiste. Por ejemplo, si se escribe por error una condición tal como *if(x = 10)* , lo que se está haciendo es asignar el valor 10 a la variable *x*, y esta orden siempre hace que la "condición" sea verdadera, porque lo que hace Arduino (tal como acabamos de explicar en el párrafo anterior) es comprobar si el valor asignado a *x* (10) es *true* (es decir, diferente de 0), cosa que es evidentemente cierto siempre. Lógicamente, lo correcto hubiera sido escribir *if(x==10)*.

Además de los operadores de comparación, en las comparaciones también se pueden utilizar los operadores booleanos (también llamados lógicos), usados para encadenar dos o más comprobaciones dentro de una condición. Son los siguientes:

Operadores lógicos

&&	Comprueba que las dos condiciones sean ciertas	(Operador AND)
 	Comprueba que, al menos, una de dos condiciones sea cierta	(Operador OR)
!	Comprueba que no se cumpla la condición a la que precede	(Operador NOT)

EL MUNDO GENUINO-ARDUINO

El operador "AND" obliga a que todas las condiciones de dentro del paréntesis del *if* sean ciertas: si hay una sola que ya no lo es, el conjunto total tampoco lo será. Al operador "OR", en cambio, le basta con que, de entre todas las condiciones, una sola ya sea cierta para que el conjunto total lo sea también. El operador NOT cambia el estado de la condición que le sigue: si esa condición era cierta pasa a ser falsa, y viceversa. El siguiente sketch ilustra lo anterior:

Ejemplo 4.31

```
void setup() {
    byte x=50;
    Serial.begin(9600);
    if (x >=10 && x <=20) {
        Serial.println("Frase 1");
    }
    if (x >=10 || x <=20) {
        Serial.println("Frase 2");
    }
    if (x >=10 && !(x<=20)) {
        Serial.println("Frase 3");
    }
}
void loop() {}
```

Si ejecutamos el código anterior, veremos que solamente se muestran la "Frase 2" y la "Frase 3". Esto es debido a que la condición del primer *if* es falsa: en él se comprueba si la variable *x* es mayor que 10 Y A LA VEZ si es menor de 20. En cambio, la condición del segundo *if* es cierta: en él se comprueba si la variable *x* es mayor que 10 O BIEN menor de 20: como ya se cumple una de las condiciones –la primera–, la condición total ya es cierta valga lo que valga el resto de condiciones presentes. La condición del tercer *if* también es cierta: en él se comprueba si la variable *x* es mayor que 10 y a la vez, gracias al operador "!", que NO sea menor de 20.

Otra funcionalidad muy útil de los *ifs* es el poder utilizar los paréntesis dentro de una condición para establecer el orden de comparación de varias expresiones. Siempre se comparan primero las expresiones situadas dentro de los paréntesis, o en el más interior en el caso de que haya varios grupos de paréntesis anidados. Por ejemplo, podríamos tener una condición como esta: *if (x!=0 || (y>=100 && y <=200))*. En ella, primero se comprueba lo que hay dentro del paréntesis, es decir, que *y* sea mayor o igual que 100 y que además sea menor o igual que 200. Eso, o que *x* sea diferente de 0. El anidamiento de condiciones puede ser todo lo complejo que uno

quiera, pero conviene asegurarse de que realmente se está comprobando lo que uno quiere, cosa que a veces, con expresiones muy largas, es difícil.

Acabaremos este apartado señalando una característica muy importante de todos los bloques condicionales (y de todos los bucles) que es la siguiente: cualquier variable declarada en el interior de un bloque/bucle es local a este. Es decir, una variable creada dentro de un bloque *if* (o *switch* o *while* o *for*, etc.) solamente existe dentro de ese bloque (y también dentro de los posibles bloques internos que pudieran existir dentro de él) pero una vez fuera de dicho bloque, esa variable se "desintegra". Es muy importante tener esto en cuenta para evitar posibles errores en nuestros sketches difíciles de localizar.

El siguiente código muestra un ejemplo de este hecho; al intentarlo compilar, el entorno Arduino nos avisará de que hay un error y de que no es posible, por tanto, dicha compilación. Ese error aparece porque la variable *x* (utilizada en la línea *Serial.println(x);* precisamente para mostrar su valor) no ha sido declarada previamente. En realidad, si que hay escrita una declaración (e inicialización) de una variable *x* pero está situada dentro de un *if*, por lo que dicha declaración solamente es tenida en cuenta dentro de ese bloque (y bloques interiores): fuera de él es como si no se hubiera escrito (aun cuando la condición del *if* sea cierta y su código interno sea ejecutado porque eso no importa: tras salir del bloque la variable deja de existir). Fijarse que este comportamiento permite la posibilidad de declarar variables homónimas dentro de bloques diferentes, pero esto en general no suele ser una práctica muy recomendable porque induce a confusión.

Ejemplo 4.32

```
void setup() { Serial.begin(9600); }
void loop() {
    if (1 < 2){
        byte x = 3;
    }
    Serial.print(x);
}
```

El bloque *switch*

Como se ha podido ver en el apartado anterior, los bloques *else if* se tienen en cuenta siempre y cuando las condiciones evaluadas hasta entonces hayan sido falsas, y la condición del propio *else if* sea la cierta. Es decir, un bloque *if(condicion1){}else if(condicion2){}* se puede leer como "si ocurre condicion1, haz el interior del primer if, y si no, mira a ver si ocurre condicion2, y (solo) si es así, haz

EL MUNDO GENUINO-ARDUINO

entonces el interior del *elseif*". Esta es una manera válida de hacer comprobaciones de condiciones múltiples, pero existe otra forma más elegante, cómoda y fácil de hacer lo mismo: utilizar el bloque *switch*. Su sintaxis es la siguiente:

```
switch (expresión) {  
    case valor1:  
        //Instrucciones que se ejecutarán cuando "expresión" sea igual a "valor1"  
        break;  
    case valor2:  
        //Instrucciones que se ejecutarán cuando "expresión" sea igual a "valor2"  
        break;  
    /*Puede haber los "case" que se deseen,  
    y al final una sección "default" (opcional)*/  
    default:  
        //Instrucciones que se ejecutan si no se ha ejecutado ningún "case" anterior  
}
```

Un bloque *switch* es como una especie de *if else* escrito más compactamente. Como se puede ver, consta en su interior de una serie de secciones *case* y, opcionalmente, de una sección *default*. Nada más llegar a la primera línea del *switch*, primero se comprueba el valor de la variable o expresión que haya entre sus paréntesis (siguiendo las mismas reglas y operadores posibles usados en un *if* estándar). Si el resultado es igual al valor especificado en la primera sección *case*, se ejecutarán las instrucciones del interior de la misma y se dará por finalizado el *switch*, continuando la ejecución del sketch por la primera línea después de la llave de cierre. En caso de no ser igual el resultado de la expresión a lo especificado en el primer *case* se pasará a comprobarlo con el segundo *case*, y si no con el tercero, etc. Por último, si existe una sección *default* (opcional) y el resultado de la expresión no ha coincidido con ninguna de las secciones *case*, entonces se ejecutarán las sentencias de la sección *default*.

En una sección *case* el valor a comparar tal solo puede ser de tipo entero. Otros lenguajes de programación permiten comparar otros tipos de datos, o comparar rangos de valores en un solo *case*, o comparar más de un valor individual en un solo *case*, etc., pero el lenguaje Arduino no.

Hay que hacer notar que una vez ejecutada una de las secciones *case* de un bloque *switch* ya no se ejecutan más secciones *case*, aunque estas también dispongan del resultado correcto de la expresión evaluada: esto es así gracias a la instrucción *break;* que comentaremos en breve.

No es necesario ordenar las secciones *case* según sus valores (de menor a mayor, o de mayor a menor), pero sí es imprescindible que la sección *default* (en caso de haberla) sea la última sección, y no puede haber más que una.

Es posible anidar sentencias *switch* sin ningún límite (es decir, se pueden poner nuevas sentencias *switch* dentro de una sección *case*).

Un código bastante sencillo que muestra el uso del bloque *switch* es este:

Ejemplo 4.33

```
void setup(){
    byte x=50;
    Serial.begin(9600);
    switch (x) {
        case 20:
            Serial.println("Vale 20 exactamente");
            break;
        case 50:
            Serial.println("Vale 50 exactamente");
            break;
        default:
            Serial.println("No vale ninguna de los valores anteriores");
    }
}
void loop(){}  


```

BLOQUES REPETITIVOS (BUCLLES)

El bloque *while*

El bloque *while* ("mientras", en inglés) es un bloque que implementa un bucle; es decir, repite la ejecución de las instrucciones que están dentro de sus llaves de apertura y cierre. ¿Cuántas veces? No hay un número fijo: se repetirán mientras la condición especificada entre sus paréntesis sea cierta (*true,1*). Su sintaxis es muy sencilla:

```
while (condición) {
    //Instrucciones que se repetirán mientras la condición sea cierta
}  


```

EL MUNDO GENUINO-ARDUINO

La condición escrita entre paréntesis sigue las mismas reglas y puede utilizar los mismos operadores que hemos visto con el bloque *if*. Nada más llegar a la línea donde aparece escrita esta condición, esta se comprobará; si resulta cierta, se ejecutarán las sentencias interiores, y si no, la ejecución del programa continuará a partir de la línea siguiente a la llave de cierre. En el primer caso (cuando la condición es cierta), una vez ejecutadas todas las instrucciones del interior del bloque *while*, se volverá a comprobar de nuevo la condición, y si esta continúa siendo cierta, se realizará otra iteración (es decir: se volverán a ejecutar las sentencias interiores). Cuando se llegue de nuevo al final de esas instrucciones anteriores, se volverá a evaluar la condición, y si sigue siendo cierta, se volverán a ejecutar. Este proceso continuará hasta que, en un momento dado, al comprobarse la condición del *while*, esta resulte falsa.

Si se llega por primera vez a una sentencia *while* y la condición resulta falsa directamente, no se ejecutarán las sentencias interiores ninguna vez. Este detalle es importante tenerlo en cuenta.

Por otro lado, las sentencias interiores a un bucle *while* pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles *while*.

En el siguiente código se puede observar cómo en el "Serial monitor" los primeros 50 mensajes que aparecerán indicarán que la variable es menor de 50 porque se está ejecutando el interior del *while*. Pero en el momento que la variable sea mayor (porque en cada iteración del bucle se le va aumentando en una unidad su valor), la condición del *while* dejará de ser cierta y saltará de allí, mostrando entonces el mensaje (ya de forma infinita) de que la variable es mayor que 50:

Ejemplo 4.34

```
byte x=1;
void setup(){
    Serial.begin(9600);
}
void loop(){
    while (x <= 50){
        Serial.println("Es menor de 50");
        x=x+1;
    }
    Serial.println("Es mayor que 50");
}
```

Una aplicación práctica del bucle *while* es el poder recibir a través del canal serie (u otro sistema de comunicación diferente) varios caracteres seguidos uno tras otro, e irlos concatenando (por ejemplo, mediante la función *unacadena.concat()*), para lograr así generar y reconocer palabras y frases enteras. Esto nos puede venir muy bien para comunicarnos con nuestra placa Arduino mediante comandos completos en vez de tan solo con simples caracteres, de tal forma que podamos construir un lenguaje de control propio (incluso con parámetros):

Ejemplo 4.35

```
char caracter;
String comando;
void setup(){
    Serial.begin(9600);
}
void loop(){
/*Voy leyendo carácter a carácter lo que se recibe por el canal serie (mientras llegue algún dato allí), y los voy concatenando uno tras otro en una cadena. En la práctica, si usamos el "Serial monitor" el bucle while acabará cuando pulsemos Enter. El delay es conveniente para no saturar el canal serie y que la concatenación se haga de forma ordenada.*/
    while (Serial.available()>0){
        caracter= Serial.read();
        comando.concat(caracter);
        delay(10);
    }
/*Una vez ya tengo la cadena "acabada", compruebo su valor y hago que la placa Arduino reaccione según sea este. Aquí podríamos hacer lo que quisieramos: si el comando es "tal", enciende un LED, si es cual, mueve un motor... y así*/
    if (comando.equals("hola") == true){
        Serial.println("El comando es hola");
    }
    if (comando.equals("adiós") == true){
        Serial.println("El comando es adiós");
    }
    //Limpiamos la cadena para volver a recibir el siguiente comando
    comando="";
}
```

Otra aplicación práctica del bucle *while* es la eliminación de los datos en el buffer de entrada del chip TTL-UART que no queramos, de manera que se quede este limpio de "basura". Esto lo podríamos hacer simplemente así:

```
while (Serial.available >0){
    Serial.read();
}
```

El bloque *do*

El bloque *do* tiene la siguiente sintaxis:

```
do {
    //Instrucciones que se repetirán mientras la condición sea cierta
} while (condición)
```

El bucle *do* funciona exactamente igual que el bucle *while*, con la excepción de que la condición es evaluada después de ejecutar las instrucciones escritas dentro de las llaves. Esto hace que las instrucciones siempre sean ejecutadas como mínimo una vez aun cuando la condición sea falsa, porque antes de llegar a comprobar esta, las instrucciones ya han sido leídas (a diferencia del bucle *while*, donde si la condición ya de entrada era falsa las instrucciones no se ejecutaban nunca).

El bloque *for*

La diferencia entre un bucle *while* (o *do*) y un bucle *for* está en que en el primero el número de iteraciones realizadas depende del estado de la condición definida, pero en un bucle *for* el número de iteraciones se puede fijar a un valor exacto. Por tanto, usaremos el bucle *for* para ejecutar un conjunto de instrucciones (escritas dentro de llaves de apertura y cierre) un número concreto de veces. La sintaxis general del bucle *for* es la siguiente:

```
for (valor_inicial_contador;condicion_final;incremento){
    //Instrucciones que se repetirán un número determinado de veces
}
```

Tal como se observa, entre paréntesis se deben escribir tres partes diferentes, separadas por los punto y coma. Estas tres partes son opcionales (pueden omitirse cualquiera de ellas) y son las siguientes:

Valor inicial del contador: en esta parte se asigna el valor inicial de una variable entera que se utilizará como contador en las iteraciones del bucle. Por ejemplo, si allí escribimos *x=0* , se fijará la variable *x* a cero al inicio del bucle. A partir de entonces, a cada repetición del bucle, esta variable *x* irá aumentando (o disminuyendo) progresivamente de valor.

Condición final del bucle: en esta parte se especifica una condición (del estilo de las utilizadas en un bucle *while*). Justo antes de cada iteración se

comprueba que sea cierta para pasar a ejecutar el grupo de sentencias internas. Si la condición se evalúa como falsa, se finaliza el bucle *for*, continuando el programa tras su llave de cierre. Por ejemplo, si allí escribimos *x<10* , el grupo interior de sentencias se ejecutará únicamente cuando la variable *x* valga menos de 10 (es decir, mientras *x<10* sea cierto).

Incremento del contador: en la última de las tres partes es donde se indica el cambio de valor que sufrirá al inicio de cada iteración del bucle la variable usada como contador. Este cambio se expresa con una asignación. Por ejemplo, la sentencia *x=x+1* le sumará 1 a la variable *x* antes de cada nueva iteración del bucle, por lo que en realidad estaríamos haciendo un contador que aumenta de uno en uno a cada repetición. Este cambio se efectúa justo antes de comprobar la condición de final del bucle.

Veamos mejor un código de ejemplo:

Ejemplo 4.36

```
byte x;
void setup(){
    Serial.begin(9600);
    for (x=0;x<10;x=x+1){
        Serial.println(x);
    }
}
void loop(){}  


```

Si observamos el resultado mostrado por el "Serial monitor", veremos que aparece una lista de números del 0 al 9. ¿Por qué? Porque cuando en el sketch se llega a la sentencia *for* primero se inicializa el contador (*x=0*) y seguidamente se comprueba la condición. Como efectivamente, *x<10*, se ejecutará la –única en este caso– sentencia interior, que muestra el propio valor del contador por el "Serial monitor". Justo después, se realiza el incremento (*x=x+1*), volviéndose seguidamente a comprobar la condición. Si *x* (que ahora vale 1) sigue cumpliendo la condición (sí, porque *1<10*), se volverá a ejecutar la instrucción interna, mostrándose ahora el valor "1" en el "Serial monitor". Justo después de esto se volverá a realizar el incremento (valiendo *x* ahora por tanto 2) y se volverá a comprobar la condición, y así hasta que llegue un momento en el que la condición resulte falsa (cuando *x* haya incrementado tanto su valor que ya sea igual o mayor que 10), momento en el cual se finalizará el *for* inmediatamente.

EL MUNDO GENUINO-ARDUINO

Es importante recalcar que el número 10 no se imprimirá, porque, tal como hemos comentado, primero se incrementa el contador y luego se hace la comprobación. Nada más acabar de ejecutar la última iteración tendremos que *x* vale 9; entonces se incrementa a 10 y seguidamente se comprueba si 10 es menor que 10. Al ser esto falso, se sale del *for*, por lo que cuando *x* vale 10 ya no tiene la oportunidad de ser imprimida. Si hubiéramos querido imprimir el 10, tendríamos que haber escrito como condición *x <=10*, en cuyo caso el *for* se hubiera repetido 11 veces.

Evidentemente, no es obligatorio que la inicialización del contador empiece por 0, ni que el incremento se realice de uno en uno (ni tan siquiera que sea un incremento: puede ser un decremento, como *x=x-1* o similar).

Como se ha mencionado, las tres partes dentro del paréntesis de la definición del bucle son opcionales. De hecho, si se omiten las tres (es decir, si se escribiera *for* (*;;*)) estaríamos escribiendo un bucle infinito, tal como la función *loop()*.

Las sentencias interiores de un bucle *for* pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles *for* (lo que se llama "for anidados").

Recordemos, por otro lado, que si se declara una variable dentro de un bucle *for*, solamente existirá mientras este bucle *for* se esté ejecutando.

Veamos otro ejemplo de uso del bucle *for*, que nos será muy útil en próximos proyectos:

Ejemplo 4.37

```
byte suma=0;
byte x;
void setup(){
    Serial.begin(9600);
    for (x=0;x<7;x=x+1){ //Sumo 7 veces el número 5
        suma= suma + 5;
    }
    Serial.println(suma);
    Serial.println(suma/7); //Esto es la media
}
void loop(){}  
}
```

En el código anterior se puede ver una manera bastante común de realizar sumas de muchos valores (y su media). Muchas veces tendremos la necesidad de calcular estas operaciones, y la mecánica siempre es la misma: en este ejemplo hemos utilizado una variable (en este caso, llamada *suma*), que empieza valiendo 0. En la primera repetición del bucle *for* se le asigna el valor del primer dato a sumar; en la segunda repetición a ese valor de *suma* se le suma el segundo valor (por lo que en ese momento *suma* vale la suma de los dos primeros valores). En la tercera repetición se le añade el tercer valor a esa suma parcial (por lo que en ese momento *suma* vale la suma de los tres primeros valores), en la cuarta repetición el cuarto valor, y así hasta que se acaba de recorrer el bucle *for* y todos los datos se han ido añadiendo uno tras otro hasta conseguir la suma total. Finalmente, muestro el resultado, y muestro también la media, que no es más que esa suma total entre el número de datos introducidos en la suma.

Otra aplicación práctica y concreta del bucle *for* es utilizarlo para recorrer los elementos de un array, uno por uno. El contador del bucle es usado como el índice (la posición) de cada elemento del array: inicialmente este contador se sitúa en el primer elemento y va aumentando en cada iteración hasta llegar al último. Por ejemplo, suponiendo que hemos declarado previamente un array de 5 elementos de tipo *char* llamado *miarray*, podríamos hacer lo siguiente para asignar un valor (el mismo, en este caso) a cada uno de ellos y seguidamente mostrarlo en el "Serial monitor" junto con su índice correspondiente:

```
for (i=0; i <5;i=i+1){  
    miarray[i]='a';  
    Serial.print(i);  
    Serial.println(miarray[i]);  
}
```

Observar en el código anterior que el contador empieza desde 0 (igual que la numeración de las posiciones de los elementos en un array) y que el bucle realizará cinco iteraciones, valiendo *i* de 0 a 4 (que corresponden con los cinco elementos del array). En cada iteración, se asigna el valor 'a' al elemento correspondiente (primero al 0, luego al 1, etc.), seguidamente se imprime el valor de *i* (0, 1, etc.) y finalmente se imprime el valor recién asignado a ese elemento (que es para todos 'a').

Por otro lado, al ser la asignación de tipo *x=x+1* muy habitual en la tercera parte de la definición del bucle *for* (y en general, en todos nuestros sketches), los programadores de Arduino suelen escribir esta asignación concreta de una manera alternativa, más compacta y rápida, mediante el uso de unos signos especiales, que pertenecen a un conjunto de símbolos llamados "operadores compuestos" cuya

EL MUNDO GENUINO-ARDUINO

función es precisamente la de facilitar la escritura (y lectura) de código. Estos operadores no son imprescindibles, porque lo que hacen se puede realizar con otros operadores (como los aritméticos: +, -, *, /, etc.) pero se utilizan mucho. A continuación, se presenta una tabla de "equivalencia" –incompleta– entre el uso de algunos de los operadores compuestos existentes en el lenguaje Arduino y su significado.

Operadores compuestos

x++	Equivale a $x=x+1$ (Al operador "++" se le llama operador "incremento")
x--	Equivale a $x=x-1$ (Al operador "--" se le llama operador "decremento")
x+=3	Equivale a $x=x+3$
x-=3	Equivale a $x=x-3$
x*=3	Equivale a $x=x*3$
x/=3	Equivale a $x=x/3$

Sabido esto, ahora podremos escribir por ejemplo un bucle *for* así: *for(x=0;x<10;x++){}}, que no es más que una forma más compacta de decir lo mismo que esto: *for(x=0;x<10;x=x+1){}**

Las instrucciones *break* y *continue*

La instrucción *break* y la instrucción *continue* están muy relacionadas con los bucles (ya sean del tipo *while* o *for*). Observar que ninguna de ellas incorpora paréntesis, pero como cualquier otra instrucción, en nuestros sketches deben ser finalizadas con un punto y coma.

La instrucción *break* debe estar escrita dentro de las llaves que delimitan las sentencias internas de un bucle, y sirve para finalizar este inmediatamente. Es decir, esta instrucción forzará al programa a seguir su ejecución a continuación de la llave de cierre del bucle. En caso de haber varios bucles anidados (unos dentro de otros), la sentencia *break* saldrá únicamente del bucle más interior de ellos.

La instrucción *continue* también debe estar escrita dentro de las llaves que delimitan las sentencias internas de un bucle y sirve para finalizar la iteración actual y comenzar inmediatamente con la siguiente. Es decir, esta instrucción forzará al programa a "volver para arriba" y comenzar la evaluación de la siguiente iteración aun cuando todavía queden instrucciones pendientes de ejecutar en la iteración

actual. En caso de haber varios bucles anidados (unos dentro de otros) la sentencia *continue* tendrá efecto únicamente en el bucle más interior de ellos. Vamos a ver ambas sentencias con un ejemplo:

Ejemplo 4.38

```
byte x;  
void setup(){  
    Serial.begin(9600);  
    for(x=0;x<10;x++){  
        if (x==4){  
            break;  
        }  
        Serial.println(x);  
    }  
}  
void loop(){}  
}
```

En el código anterior, se puede ver cómo el valor 4 de x ya no se llega a imprimir porque la instrucción *break* interrumpe la ejecución del bucle entero. Como tras el *for* la función *setup()* ya no tiene más código, no se ve nada más.

¿Qué pasaría si sustituimos la instrucción *break* por la sentencia *continue*, dejando el resto del código anterior exactamente igual? Que veríamos una lista de números desde el 0 hasta el 9, excepto precisamente el 4. Esto es así porque la instrucción *continue* interrumpe la ejecución de la iteración en la cual x es igual a 4 (y por tanto, la instrucción *Serial.println()* correspondiente no se llega a ejecutar) pero continúa con la siguiente iteración del bucle de forma normal, en la cual a x se le asigna el valor 5.

5 LIBRERÍAS ARDUINO

LAS LIBRERÍAS OFICIALES

Ya sabemos que una librería sirve para añadir al lenguaje Arduino base un determinado conjunto de instrucciones extra que ofrecen cierta funcionalidad suplementaria (la cual puede ser muy diferente dependiendo de la librería en particular que se esté utilizando: hay librerías que ofrecen, por ejemplo, la posibilidad de gestionar protocolos de comunicación diversos, otras que permiten interactuar con hardware variado de una forma más cómoda, otras que facilitan la manipulación de datos en diferentes formatos, etc.).

A continuación, se hará un breve resumen de las librerías oficiales existentes. Al contrario de lo que ocurre con las librerías de terceros (que para poder ser reconocidas por el IDE de Arduino, recordemos, han de ser importadas previamente mediante la opción "Sketch"->"Include library"->"Add .ZIP library" del menú del IDE o bien mediante el "Library Manager" accesible a través de la opción "Sketch"->"Include library"->"Manage libraries..."), las librerías oficiales siempre se instalan junto con el IDE y, por tanto, ya aparecen automáticamente listadas bajo el menú "Sketch->"Include library". Esto significa que, para poder empezar a usar cualquiera

EL MUNDO GENUINO-ARDUINO

de las librerías oficiales, tan solo debemos seleccionarla de dicha lista y nada más; al realizar esta selección, aparecerá en nuestro sketch una referencia a esa librería en forma de una –o más– líneas `#include` (líneas que, por otro lado, pueden ser escritas a mano si así lo quisieramos), lo que nos indica que la funcionalidad de dicha librería ya está disponible para ser empleada en nuestro sketch.

Librería LiquidCrystal

Permite controlar todas las pantallas de cristal líquido (LCDs) que estén basadas en el chip HD44780 de Hitachi (o compatibles, como el KS0066 de Samsung o el ST7065C de Sitronix, entre otros). Estos modelos de chip (presentes en la gran mayoría de LCDs de caracteres del mercado) son capaces de trabajar tanto en modo 4-bit como en 8-bit (es decir: pueden utilizar tanto 4 como 8 líneas de datos); la librería "LiquidCrystal" es compatible con ambos modos de trabajo. Estudiaremos su uso en un apartado posterior de este capítulo.

Librería SD

Permite leer/escribir datos de/en una tarjeta SD (o microSD) conectada a un zócalo compatible, ya esté este presente en algún shield (como por ejemplo el Arduino Ethernet Shield) o en algún módulo de terceros. Las tarjetas SD son muy útiles para almacenar ficheros tales como audio, vídeo o imágenes, o bien datos textuales obtenidos de diferentes sensores, ya que ofrecen un sistema de grabación mucho mayor que la memoria EEPROM.

La librería SD puede utilizar tarjetas tanto de tipo SDSC como de tipo SDHC, y puede trabajar con tarjetas formateadas en los sistemas de ficheros FAT16 y FAT32. La comunicación entre la tarjeta SD y el microcontrolador se establece internamente usando el protocolo SPI. Estudiaremos el uso de esta librería en un apartado posterior de este capítulo.

Librería Ethernet

Permite conectar el Arduino Ethernet Shield (u otros shields/placas similares, algunos de los cuales estudiaremos en el capítulo 8) a una red Ethernet (TCP/IP). Se puede configurar para que la placa actúe como servidor (es decir, que de forma permanente e ininterrumpida escuche y acepte peticiones de otros dispositivos de la red que soliciten algún tipo de servicio o dato ofrecido por ella) o bien como cliente (es decir, que sea la placa la que solicite puntualmente esos servicios o datos a otro dispositivo de red). Esta librería soporta hasta un total cuatro conexiones concurrentes (entrantes –es decir, en "modo servidor"–, salientes –es decir, en "modo cliente"– o una combinación de estas).

Librería WiFi101

Permite conectar el Arduino WiFi Shield 101 (u otros similares, algunos de los cuales estudiaremos en el capítulo 8) a una red WiFi (TCP/IP). Al igual que ocurre con la librería "Ethernet", la librería "WiFi101" permite que el WiFi Shield 101 actúe como servidor o bien como cliente.

Librería Temboo

Permite comunicar cualquier placa Arduino que sea capaz de conectar con redes TCP/IP (como es por ejemplo el modelo Yún u otros si vienen acompañados del Arduino Ethernet Shield o del Arduino WiFi Shield 101) con la plataforma online Temboo (<https://www.temboo.com/arduino>), la cual ofrece al hardware Arduino la posibilidad de interactuar de una forma centralizada y homogénea con decenas de servicios disponibles en Internet de todo tipo: bases de datos, servidores de correo, servidores de noticias, etc. etc. En otras palabras: Temboo permite que nuestra placa Arduino pueda recibir información proveniente de múltiples lugares de Internet convenientemente filtrada (incluyendo no solo datos presentes en distintos portales web sino también la recepción de órdenes de control enviadas desde cualquier navegador), así como que pueda aportar automáticamente datos generados por ella misma a multitud de destinos online diferentes (Twitter, Facebook, etc.).

Librería GSM

Permite conectar el Arduino GSM Shield (u otros de características similares) a la red de telefonía móvil digital GSM/GPRS. Gracias a ello, la placa Arduino acoplada a él podrá realizar la mayoría de operaciones que se esperan de un teléfono de segunda generación, como efectuar y recibir llamadas de voz, enviar y recibir mensajes SMS/MMS o conectar a Internet de forma básica.

Librería SPI

Permite comunicar mediante el protocolo SPI la placa Arduino (que actuará siempre como "maestro") con dispositivos externos (que actuarán siempre como "esclavos"). Tal como ya se ha comentado, la comunicación se establece mediante el pin MOSI (que implementa la línea de envío de datos del maestro al esclavo), el MISO (que implementa la línea de envío de datos del esclavo al maestro), el SCK (que implementa la línea de reloj) y el pin SS (que implementa la línea de selección del esclavo); los tres primeros pines están ubicados en el conector ICSP de cualquiera de los modelos de placas Arduino y el último puede ser cualquier pin de salida digital.

EL MUNDO GENUINO-ARDUINO

Debido a que esta librería es capaz de controlar hasta el mínimo detalle el proceso de comunicación establecido con los periféricos SPI, es relativamente compleja de utilizar ya que se necesitan ciertos conocimientos avanzados de electrónica para poder aprovecharla en todas sus posibilidades. ¿Esto quiere decir que no podremos utilizar dispositivos SPI en nuestros proyectos? ¡No! Afortunadamente, existen muchas librerías de terceros que ocultan la complejidad interna de la librería SPI y ofrecen al usuario una serie de instrucciones mucho más sencillas de aprender y utilizar. Los dos únicos pequeños inconvenientes de estas librerías "para novatos" (llámémoslas así) es que lógicamente no ofrecen toda la flexibilidad y capacidad que ofrece la librería SPI directamente, y que normalmente las librerías "para novatos" están enfocadas para un tipo determinado de dispositivo SPI muy concreto. Esto último implica que para cada dispositivo SPI deberemos de utilizar una librería "para novatos" diferente (cuando todos ellos en realidad se manejan por debajo con una única librería, la SPI). Además, corremos el riesgo de que para ese dispositivo en concreto que queremos usar nadie haya desarrollado su librería "para novatos" correspondiente aún. No obstante, a pesar de todo esto, para lograr códigos más comprensibles, en este libro se ha optado por ver diversos ejemplos de manejo de dispositivos SPI haciendo uso de sus librerías particulares.

Librería Wire

Permite comunicar mediante el protocolo I²C (también llamado TWI) la placa Arduino con dispositivos externos. Tal como ya se ha comentado varias veces, la comunicación se establece a través del pin SDA (que implementa la línea de datos) y el pin SCL (que implementa la línea de reloj); ambos pines solo se utilizan para este menester. Por otro lado, esta librería utiliza 7 bits para identificar el dispositivo (lo que da la posibilidad de distinguir hasta 128 dispositivos diferentes).

Al igual que la librería SPI, la librería Wire es bastante compleja porque permite controlar hasta el mínimo detalle el proceso de comunicación entre dispositivos. Es por eso que en este libro se optará por utilizar librerías "para novatos" que ocultan los entresijos internos del proceso comunicativo y ofrecen al usuario una forma mucho más sencilla, clara y rápida de escribir sus sketches. Normalmente, cada dispositivo diferente usará su librería "para novatos" correspondiente, por lo que nos encontraremos con bastante variedad de librerías para elegir según el dispositivo que queramos usar, a pesar de que todas ellas se basen en una única librería, la Wire.

Librería SoftwareSerial

Ya sabemos que gracias al chip TTL-UART de la mayoría de placas Arduino, estas son capaces de comunicarse con dispositivos externos estableciendo una

conexión serie a través de (al menos) los pines 0 (RX) y 1 (TX). La librería SoftwareSerial lo que permite es que este tipo de comunicación se pueda establecer con dispositivos conectados a pines cualesquiera, diferentes del 0 y del 1 (o diferentes de los otros pines que, como es el caso del modelo Mega, estén diseñados por hardware para gestionar ese tipo de comunicación). Por eso esta librería se llama así: porque consigue simular un chip TTL-UART "virtual" mediante software.

Gracias a esta librería, es posible tener múltiples puertos serie con una velocidad de transferencia de hasta 115200 bits/s. Y al igual que el chip TTL-UART "real", cada puerto serie "simulado" dispone de un buffer de 64 bytes, de tal manera que el microcontrolador pueda acumular allí los datos recibidos mientras está trabajando en otras cosas. No obstante, existen varias limitaciones: la más importante es que, si se utilizan varios puertos, solamente uno podrá recibir datos a la vez. Estudiaremos el uso de esta librería en un apartado posterior de este capítulo.

Librería Firmata

Permite comunicar a través de una conexión existente (ya sea de tipo serie vía cable USB –u otros medios–, o vía cable Ethernet, o vía WiFi, o vía Bluetooth, etc.) una placa Arduino con programas ejecutados en un computador, de forma que estos puedan controlar remotamente esa placa a la vez que puedan recibir de ella los datos pertinentes en cada momento.

Firmata (<https://github.com/firmata/arduino>) es un protocolo genérico de comunicación entre microcontroladores y software informático que permite la transferencia de órdenes y datos entre ambos de una forma muy cómoda. Firmata requiere que en la placa Arduino se esté ejecutando un determinado sketch que haga uso de la librería homónima (el cual muchas veces no será necesario ni que lo desarrollemos nosotros mismos porque bastará con utilizar alguno de los sketches prefabricados que vienen incluidos en el IDE bajo el menú de ejemplos, como es el caso del popular "StandardFirmata") y que, además, en el computador se esté ejecutando una aplicación compatible con dicho sketch; esta aplicación puede estar desarrollada en cualquiera de los lenguajes de programación que dispongan de la capacidad de tratar con el protocolo Firmata, los cuales son la mayoría (Processing, Python, Javascript, Java, .NET, Php, etc.); un ejemplo de aplicación de este estilo es, por ejemplo "Snap4Arduino", mencionada en el capítulo 3. Esta librería no será estudiada en este libro pero si se desea aprender sobre ella, se puede consultar su página oficial y también la documentación de referencia de la página oficial de Arduino: <http://arduino.cc/en/Reference/Firmata>.

Librerías Servo y Stepper

La librería Servo sirve para facilitar al programador de la placa Arduino el control de servomotores. Concretamente, permite manejar hasta 12 servomotores en la placa Arduino UNO (y hasta 48 en la Arduino Mega!). Solo hay que tener en cuenta que, en el caso de emplear una placa basada en el microcontrolador ATmega328P, la simple declaración de esta librería en nuestro sketch (aunque no haya físicamente ningún servomotor conectado a nuestro circuito) monopoliza el "Timer1" de ese microcontrolador (y, por tanto, deshabilita las demás funcionalidades dependientes de él, como es la posibilidad de generar señales PWM en los pines de la placa asociados –recordemos que en el caso del modelo UNO, estos son los nº 9 y nº 10–).

La librería Stepper, por su parte, sirve para controlar motores tipo "paso a paso" (en inglés, "steppers"), tanto de tipo unipolar como bipolar. Estudiaremos el uso de ambas librerías en un apartado posterior de este capítulo.

Librerías Keyboard y Mouse (solo para placas basadas en el chip ATmega32U4 y para los modelos Due y Zero)

La librería Keyboard permite a las placas mencionadas en el título actuar, cuando son conectadas a un computador, como si fueran un teclado (donde las pulsaciones de teclas serán simuladas mediante la interacción con diferentes elementos de nuestro circuito, como pulsadores y/o sensores, por ejemplo).

La librería Mouse, por su parte, permite a las mismas placas actuar como si fueran un ratón, pudiendo controlar (también mediante la interacción con componentes presentes en nuestro circuito) el movimiento del cursor en la pantalla del computador conectado a la placa en cuestión. Estudiaremos el uso de ambas librerías en un apartado específico del capítulo 6.

Las dos librerías se basan en un código común genérico llamado HID (de "Human Interface Device") que está encapsulado en forma de librería suplementaria y viene instalada por defecto con el IDE. Esta librería no suele ser utilizada directamente en los sketches pero siempre es importada automáticamente junto con las propias Keyboard y/o Mouse para que estas últimas funcionen de forma correcta.

Conviene destacar la existencia de otra librería también llamada HID (no oficial) disponible en <https://github.com/NicoHood/HID>. Esta librería tiene la interesante característica de incorporar (en forma de librerías derivadas) la definición de otros muchos dispositivos de interacción con el ordenador distintos además de los oficiales Keyboard y Mouse, como por ejemplo diversos mandos de consola (librería "Gamepad"), aparatos MIDI (librería "Midi"), etc.

Por otro lado, también es interesante saber que es posible ejecutar sketches en los modelos UNO y Mega que hagan uso de las librerías Keyboard y Mouse (así como también del resto de librerías derivadas de la HID no oficial mencionada en el párrafo anterior). Para ello, no obstante, primero deberemos sobrescribir el bootloader del chip ATmega16U2 por otro gestor de arranque diferente llamado HoodLoader2 (<https://github.com/NicoHood/HoodLoader2>; el procedimiento de escritura a seguir está explicado de forma genérica en el apartado del capítulo 2 que trata sobre la reprogramación DFU del ATmega16U2, pero para conocer los detalles exactos sobre cómo escribir este bootloader concreto es conveniente consultar su web). El programa HoodLoader2 tiene la peculiaridad de permitir cargar (y ejecutar) sketches tanto en el chip ATmega328P (lo que viene siendo lo habitual) como en el chip ATmega16U2, el cual no deja de ser una versión algo más limitada del ATmega32U4. Por tanto, cargando un sketch en el ATmega16U2 dispondremos de prácticamente toda la funcionalidad de una placa Arduino basada en Atmega32U4, entre la que se encuentra, precisamente, la posibilidad de utilizar las librerías Keyboard y Mouse. Siguiendo las instrucciones marcadas en <https://github.com/NicoHood/HoodLoader2/wiki/How-to-use-reset> podremos elegir, en el momento de arrancar nuestra placa, cuál de los dos sketches (el residente en el ATmega328P o en el ATmega16U2) queremos poner en marcha.

Librería EEPROM (para todas las placas excepto los modelos Due y Zero)

Permite leer y escribir datos en la memoria EEPROM del microcontrolador. Recordemos que este tipo de memoria solamente está presente en aquellas placas Arduino cuyo microcontrolador es de tipo AVR, por lo que esta librería no funcionará si usamos los modelos Due y Zero –ya que son de tipo ARM–. Estudiaremos su uso en un apartado posterior de este capítulo.

Recordemos también que la característica principal de una memoria EEPROM es que puede mantener grabados los datos aunque la placa deje de recibir alimentación eléctrica, se resetee o se sobrescriba el sketch a ejecutar. Su mayor limitación, por otro lado, es la cantidad de veces que se pueden leer o escribir datos en ella: según Atmel, la EEPROM del ATmega328P es capaz de soportar "solo" hasta 100000 ciclos de lectura/escritura.

Librerías USBHost y Scheduler (solo para los modelos Due y Zero)

La librería USBHost permite a nuestros sketches actuar como "hosts" USB, de manera que diferentes periféricos USB (tales como teclados o ratones, entre otros) se puedan conectar al zócalo micro-USB "nativo" de la placa Arduino Due o Zero para así interactuar directamente con ella.

EL MUNDO GENUINO-ARDUINO

Concretamente, para gestionar la conexión de un teclado (es decir, básicamente para reconocer qué teclas son pulsadas) deberemos emplear dentro de nuestro sketch funciones asociadas al objeto *KeyboardController* perteneciente a esta librería; asimismo, para gestionar la conexión de un ratón (es decir, básicamente para detectar qué botón ha sido o está siendo clicado, o para saber si el ratón está siendo movido/arrastrado y la magnitud de ese desplazamiento) deberemos emplear las funciones asociadas al objeto *MouseController*. Esta librería no será estudiada en este libro, pero si el lector desea obtener más información, puede consultar la referencia oficial en <https://www.arduino.cc/en/Reference/USBHost>.

La librería Scheduler, por su parte, permite a nuestros sketches realizar múltiples tareas al mismo tiempo sin que se bloqueen entre sí, tal como si el sketch estuviera compuesto por varias funciones *loop()* independientes ejecutándose en paralelo. Esta librería no será estudiada en este libro, pero si el lector desea obtener más información, puede consultar su referencia oficial (disponible en <https://www.arduino.cc/en/Reference/Scheduler>) donde podrá ver que esta librería sólo consta de dos funciones: *Scheduler.startLoop()* y *yield()*).

Librería Audio (solo para el modelo Due)

La librería Audio permite a nuestros sketches reproducir (a través del circuito auxiliar adecuado, formado como mínimo por un amplificador y un altavoz) ficheros de audio en formato WAV almacenados en una tarjeta SD. Esto es posible gracias a los dos conversores digital-analógicos (DAC0 y DAC1) que incorpora la placa Due, los cuales posibilitan tener dos salidas analógicas reales de 12 bits de resolución (es decir, de hasta 4096 niveles diferentes) con las que se puede generar audio de calidad. Esta librería no será estudiada en este libro, pero si el lector desea obtener más información, puede consultar su referencia oficial (disponible en <https://www.arduino.cc/en/Reference/Audio>) donde podrá ver que esta librería solo consta de tres funciones: *Audio.begin()*, *Audio.prepare()* y *Audio.write()*.

Librerías AudioZero y RTCZero (solo para el modelo Zero)

La librería AudioZero permite a nuestros sketches reproducir (a través del circuito auxiliar adecuado, formado como mínimo por un amplificador y un altavoz) ficheros de audio en formato WAV almacenados en una tarjeta SD. Esto es posible gracias al conversor digital-analógico (DAC0) que incorpora la placa Zero, el cual posibilita tener una salida analógica real de 10 bits de resolución (es decir, de hasta 1024 niveles diferentes) con los que se puede generar audio de calidad.

La librería RTCZero permite controlar el RTC ("Real Time Clock") que hay en la placa Arduino Zero. Un RTC es un reloj que lleva el seguimiento de la hora y fecha actual (con precisión de hasta segundos) de manera que puede ser usado, por ejemplo, para programar la realización de tareas en determinados momentos (a modo de "alarmas") o para llevar el registro de cada instante en el que se recibe algún dato de un sensor, etc. El RTC en concreto de la placa Arduino Zero se basa en un oscilador de cristal de cuarzo, funciona a 32768 Hz, frecuencia que casualmente es igual a 2^{15} , lo cual hace que sea muy conveniente en circuitos contadores binarios; otra característica interesante de este RTC es que puede seguir funcionando aunque la placa Zero esté en el modo de bajo consumo, de manera que puede ser usado, si así lo deseamos, para "despertar" dicha placa de forma programada (no obstante, lógicamente, si dicha placa dejara de recibir alimentación, el RTC también dejaría de funcionar –y, por tanto, perdería la referencia con la fecha y hora reales–; para evitar este problema lo más recomendable es utilizar alguna fuente de alimentación suplementaria –como pueden ser pilas botón– para mantener al menos el circuito RTC activo de forma permanente).

Ninguna de estas dos librerías será estudiada en este libro, pero si el lector desea obtener más información, puede consultar la referencia oficial, disponible respectivamente en <https://www.arduino.cc/en/Reference/ArduinoZero> (allí se puede ver que esta librería solo consta de tres funciones: `AudioZero.begin()`, `AudioZero.play()` y `AudioZero.end()`) y <https://www.arduino.cc/en/Reference/RTC> (allí se puede ver que esta librería consta de muchas funciones –todas ellas asociadas a un objeto de tipo `RTCZero` que podremos llamar, por ejemplo, `rtc`–, pero muy sencillas de utilizar y comprender: además de `rtc.begin()` –imprescindible para poder activar el RTC de la placa–, nos ofrece funciones para establecer la fecha y la hora –`rtc.setTime()` o `rtc.setDate()`, entre otras–, funciones para obtener la fecha y la hora actual –`getSeconds()`, `getMinutes()`, `getHours()`, `getDay()`, `getMonth()` y `getYear()`– o funciones para definir una alerta, es decir, para establecer una fecha/hora en la que se disparará una determinada acción: `setAlarmTime()`, `setAlarmDate()`, `attachInterrupt()`, `detachInterrupt()`, `enableAlarm()` y `disableAlarm()`).

Librerías Bridge y SpacebrewYún (solo para el modelo Yún)

La librería Bridge permite comunicar el microcontrolador ATmega32U4 de la placa Arduino Yún con el sistema Linux ejecutado por el chip Atheros 9331. Básicamente, permite que un sketch Arduino pueda utilizar los recursos ofrecidos por ese sistema Linux para, por ejemplo, acceder a Internet gracias a la configuración de red (Ethernet o WiFi) de dicho sistema y recibir las respuestas correspondientes, y/o para navegar por la jerarquía de carpetas y ficheros de dicho sistema (ubicada en una

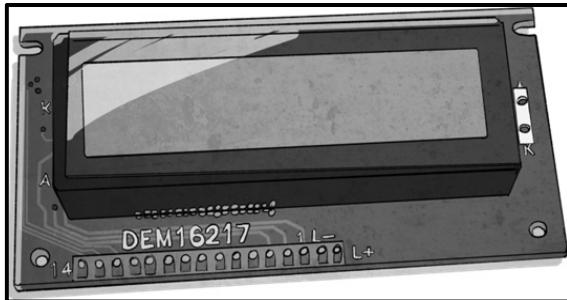
EL MUNDO GENUINO-ARDUINO

tarjeta SD) y así poder obtener (o almacenar) los datos pertinentes, y/o para ejecutar programas de terminal (previamente instalados en dicho sistema) y recibir los resultados oportunos, etc., etc. Esta librería, debido a su relativa complejidad, no será estudiada en este libro.

La librería SpacebrewYún, por su parte, permite conectar la placa Arduino Yún con un determinado servicio online llamado Spacebrew (<http://docs.spacebrew.cc>), cuyo objetivo es comunicar directamente diferentes placas (y en general, objetos interactivos que tengan conexión de red) entre sí a través de Internet (o a través de nuestra propia red local si descargáramos e instaláramos en un computador propio el software servidor de Spacebrew –ya que este proyecto es de código libre y multiplataforma–). En este libro no la veremos debido a su poca relevancia actual.

USO DE PANTALLAS LCD

Las pantallas de cristal líquido (LCDs)



Las pantallas de cristal líquido (en inglés "Liquid Crystal Displays" –LCDs–) ofrecen una manera muy rápida y vistosa de mostrar mensajes. Las podemos clasificar en LCDs de caracteres y LCDs gráficas (estas últimas también llamadas GLCDs). Las primeras sirven para mostrar texto ASCII y

se comercializan en diferentes tamaños (16x2, 20x4...) donde el primer número indica la cantidad de caracteres que caben en una fila, y el segundo número es el número de filas que caben en la pantalla. Las segundas sirven para mostrar, además de texto, dibujos e imágenes, y también se comercializan en diferentes tamaños, los cuales están definidos por la cantidad de píxeles –es decir, puntos de luz– que pueden mostrar (128x64, 128x128...). Las LCDs de caracteres, por su parte, pueden mostrar pequeños iconos de 5x7 píxeles o similar.

Las LCDs de caracteres más habituales están basadas en el chip HD44780 de Hitachi (o compatibles como el KS0066 de Samsung o el ST7065 de Sitronix, entre otros) y ofrecen 16 pines de conexión. De estos, tan solo cuatro u ocho (según si el LCD esté trabajando en modo "4-bit" o "8-bit", respectivamente) son utilizados para transferencia de datos (bits). Esto no significa, sin embargo, que solamente se

necesiten cuatro (u ocho) cables para conectar el LCD a nuestro circuito, ya que además de recibir o enviar datos, la pantalla deberá conectarse a la alimentación (mediante otro cable), a tierra (con otro cable más), a la línea de reseteado (con otro más), etc. Además, hay que tener en cuenta otras características que ofrecen la mayoría de LCDs de caracteres (y que requieren sendos cables extra), como son la posibilidad de iluminar el fondo de la pantalla (opción ideal para entornos con poca luz ambiental) o la de utilizar varios colores de fondo (y no solamente el blanco/negro sobre azul/verde tradicional), etc. En la práctica, cada modelo de LCD es diferente, por lo que será imprescindible consultar su datasheet concreto para poder confirmar cuántos pines consta ese LCD en particular, qué función realizan y qué ubicación tienen. De todas formas, a continuación detallamos los pines más habituales ofrecidos por cualquier LCD estándar:

Un pin para recibir la alimentación (normalmente con los 5V que proporciona la placa Arduino ya está bien, pero hay modelos que requieren 3,3V, así que cuidado) **y otro pin para conectar la pantalla a tierra.**

NOTA: Es conveniente conectar un divisor de tensión entre la fuente de alimentación y el pin de alimentación de la pantalla para evitar posibles daños. Para calcular el valor óptimo de esta resistencia, se deben consultar dos valores en el datasheet del LCD: la corriente máxima soportada para la luz de fondo y la caída de tensión causada por esta. Haciendo uso de la Ley de Ohm, si se resta dicha caída de tensión de los 5V y se divide el resultado entre esa corriente máxima, obtendremos el valor de la resistencia (redondeando al alza) que necesitamos. Por ejemplo, si la corriente máxima es de 16mA y la caída de tensión es de 3,5V, la resistencia debería ser $(5 - 3,5)/0,016 = 93,75$ ohmios (o 100 ohmios redondeando a un valor estándar). Si no se puede consultar el datasheet, un valor seguro para usar son 220 ohmios, aunque un valor tan alto hará que la luz de fondo sea más tenue.

Un pin para regular el contraste de la pantalla. Este pin se debe conectar a la patilla central de un potenciómetro de nuestro circuito (el cual a su vez ha de tener sus patillas exteriores conectadas a la alimentación y tierra, respectivamente), de manera que regulando el potenciómetro podremos regular el contraste de la pantalla para tener una mejor visión (el valor concreto elegido dependerá del ángulo de visión que tengamos, de la iluminación ambiente, etc). Alternativamente, este pin también podría conectarse directamente a una salida PWM de nuestra placa Arduino para que el control del contraste se pudiera hacer de forma programada en vez de manualmente.

Tres pines de control generalmente marcados como "RS", "EN" y "RW", que se deberán conectar cada uno a un pin digital de la placa Arduino. El pin "RS"

EL MUNDO GENUINO-ARDUINO

sirve para que el microcontrolador le diga a la LCD si quiere mostrar caracteres o si lo que quiere es enviar comandos de control (como cambiar la posición del cursor o borrar la pantalla por ejemplo). Concretamente, si por ese pin el LCD detecta una señal LOW, los datos recibidos serán tratados como comandos a ejecutar, y si detecta una señal HIGH, los datos recibidos serán el texto a mostrar en la pantalla. El pin "EN" establece la línea "enable", la cual sirve para advertir a la LCD que el microcontrolador le va a enviar datos (ya sean de control o para imprimir). Esta advertencia se produce cada vez que la señal recibida por ese pin cambia de HIGH a LOW. Finalmente, el pin "RW" sirve para definir si se desea enviar datos a la LCD (lo más común) o recibirlos de ella (muy poco común); si estamos en el primer caso, este pin deberá recibir una señal LOW y en el segundo caso deberá recibir una señal HIGH, por lo que, como normalmente no lo necesitaremos para nada, en la práctica lo conectaremos casi siempre a tierra.

Varios pines (4 u 8, según si la LCD funciona en modo "4-bit" o "8-bit") que se deberán conectar también cada uno a un pin digital de la placa Arduino. Se usan para establecer las líneas de comunicación en paralelo por donde se transfieren los datos (bits) y los comandos de control de la placa Arduino hacia el LCD. En este sentido, hay que tener en cuenta que, aunque en modo "4-bit" se requieran la mitad de cables, la velocidad de transferencia de información es mucho menor que en modo "8-bit" (concretamente, en una relación aproximada de 14 a 1, según modelos).

Dos pines exclusivos para el circuito de la luz de fondo (uno para recibir la alimentación -la cual puede atenuarse mediante la conexión en serie de un divisor de tensión o también mediante una señal PWM- y el otro pin conectado a tierra). Si la pantalla no dispone de luz de fondo (también llamada de "retroalimentación"), estos pines o no existen o no son usados para nada.

Algunos modelos de LCD (no todos) disponen además de una característica llamada RGB (de "Red-Green-Blue") que consiste en ofrecer la opción de cambiar el color de la luz de fondo. Esto es posible gracias a la existencia dentro de la pantalla LCD de tres LEDs (cada uno de un color primario –rojo, verde y azul–) asociados a **tres pines extra más**, los cuales deberán conectarse a sendos pines de salida PWM de la placa Arduino.

Como guía para el lector, podemos nombrar como ejemplo de pantallas LCD de caracteres estándar los productos siguientes de Adafruit (todos vienen con

potenciómetro incluido para el control de contraste): el nº **181** (de 2x16, caracteres blancos sobre fondo azul), el nº **198** (de 4x20, blanco sobre azul), el nº **399** (2x16, caracteres de colores RGB sobre fondo negro) o el nº **398** (2x16, caracteres negros sobre fondo RGB). Sparkfun por su parte también distribuye, entre otros modelos, el producto nº **255** (2x16,negro sobre verde), el nº**256** (4x20, negro sobre verde), el producto nº **11122** (2x8, negro sobre verde), el nº **709** (2x16, blanco sobre negro), el nº **790** (2x16, amarillo sobre azul), el nº **791** (2x16, rojo sobre negro) o el nº **10862** (2x16, caracteres negros sobre fondo RGB). Todos ellos son compatibles con la librería "LiquidCrystal" oficial de Arduino y funcionan a 5V. Si se desean más modelos, se pueden buscar en el distribuidor especializado <http://www.tinsharp.com>.

La librería LiquidCrystal

Lo primero que debemos hacer para poder utilizar pantallas LCD compatibles con la librería oficial "LiquidCrystal" es declarar una variable global de tipo *LiquidCrystal*, la cual representará dentro de nuestro sketch al objeto LCD que queremos controlar. La declaración se ha de realizar usando la siguiente sintaxis (suponiendo que llamamos *milcd* a dicha variable-objeto): *LiquidCrystal milcd(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7)*; donde todos los parámetros especificados entre paréntesis en realidad son valores numéricos que representan:

rs : nº del pin de la placa Arduino conectado al pin "RS" de la LCD.

rw: nº del pin de la placa Arduino conectado al pin "RW" de la LCD. Opcional.

enable: nº del pin de la placa Arduino conectado al pin "ENABLE" de la LCD.

d0... hasta d7: números de los pines de la placa Arduino conectados a los pines de datos correspondientes de la LCD. Los parámetros *d0*, *d1*, *d2* y *d3* son opcionales: si se omiten, la LCD será controlada usando solo cuatro líneas (*d4*, *d5*, *d6*, *d7*) en vez de 8 (es decir, funcionará en modo "4-bit" en vez de "8-bit").

Por ejemplo, una posible declaración para una LCD de 4 bits sin uso del pin "RW" podría ser: *LiquidCrystal milcd(12, 11, 5, 4, 3, 2);*

Una vez creado el objeto *milcd* con la línea anterior, lo primero que debemos hacer es establecer el tamaño de la pantalla para poder trabajar con ella. Esto se hace mediante la siguiente función:

milcd.begin(): especifica las dimensiones (columnas y filas) de la pantalla. Tiene dos parámetros: el primero es el número de columnas que tiene la pantalla y el segundo es el número de filas. No tiene valor de retorno. Esta función ha de ser ejecutada antes de poder empezar a imprimir ningún carácter en ella.

EL MUNDO GENUINO-ARDUINO

A partir de aquí, ya podremos escribir caracteres en la pantalla con alguna de las tres funciones siguientes:

milcd.write(): escribe un carácter en la pantalla. Como único parámetro tiene ese carácter, el cual se puede especificar explícitamente entre comillas simples o bien a través de una variable de tipo *byte* (cuyo valor puede haber sido obtenido de un sensor, o leído del canal serie, etc.). Su dato de retorno es de tipo *byte* y vale el número de bytes escritos (por tanto, si todo es correcto, valdrá 1), aunque no es obligatorio utilizarlo.

milcd.print(): escribe un dato (de cualquier tipo) en la pantalla. Como primer parámetro tiene ese dato, que puede ser tanto un carácter de tipo *char* como una cadena de caracteres, pero también puede ser numérico entero (*int*, *long*, etc). Opcionalmente, se puede especificar un segundo parámetro que indicará, en el caso de que el dato a escribir sea entero, el formato con el que se verá: puede valer la constante predefinida *BIN* (que indicará que el número se visualizará en formato binario), *HEX* (en hexadecimal) y *DEC* (en decimal, que es su valor por defecto). Su dato de retorno es de tipo *byte* y vale el número de bytes escritos, aunque no es obligatorio utilizarlo.

milcd.createChar(): crea un carácter totalmente personalizado. Se pueden crear hasta ocho caracteres diferentes de 5x8 píxeles cada uno. La apariencia de cada carácter es especificada por un array de 8 bytes. Cada uno de estos bytes representa una fila de píxeles. Cada fila de píxeles se dibuja según los valores de los últimos cinco bits individuales (los de más a la derecha) de su byte correspondiente: si el bit vale 1, se pintará el píxel pertinente y si vale 0, no. No obstante, esta función no escribe el carácter en la pantalla, tan solo lo crea; para poderlo escribir, deberemos utilizar *milcd.write()*. Esta función tiene dos parámetros: el primero es la identificación del carácter a crear (puede ser un número entre 0 y 7) y el segundo es el array con el "dibujo" de los píxeles. Para transformar el dibujo que queremos en los valores adecuados del array, podemos usar la ayuda interactiva que ofrece la página <http://www.quinapalus.com/hd44780udg.html> o también <http://omerk.github.io/lcdchargen>.

Veamos un ejemplo de uso de las funciones anteriores. Suponiendo que tenemos ya conectado el pin *rs* del LCD al pin nº 12 de la placa Arduino, el pin *enable* al pin número 11 y los cuatro pines de datos *d4*, *d5*, *d6* y *d7* a los pines 5, 4, 3 y 2, (además de tener conectados también como mínimo los pines de alimentación y tierra del LCD), el siguiente código mostraría un ícono sonriente en la pantalla:

Ejemplo 5.1

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
/*Para poder establecer individualmente los valores de todos los bits que forman cada byte del array
(en otras palabras, para poder escribir el contenido de esos bytes en sistema binario), recordemos que el
valor de cada byte se ha de preceder con el signo 'B'. */
byte smiley[8] = { B00000, //En sistema decimal los mismos valores de los bytes son: 0,17,0,0,17,14,0
                    B10001,
                    B00000,
                    B00000,
                    B10001,
                    B01110,
                    B00000,
                    B00000 };

void setup() {
    milcd.createChar(0, smiley);
    milcd.begin(16, 2); //La pantalla es de 2 filas de 16 caracteres
    milcd.write(byte(0)); //Importante hacer notar la necesidad de realizar un 'casting' a byte
}
void loop() {}
```

A partir de aquí, podemos manipular la pantalla de diferentes formas con la ayuda de las instrucciones siguientes:

milcd.clear(): borra la pantalla y posiciona el cursor en su esquina superior izquierda para escribir a partir de allí el próximo texto. No tiene ni parámetros ni valor de retorno.

milcd.home(): posiciona el cursor en la esquina superior izquierda de la pantalla para escribir a partir de allí el próximo texto. Si además se quiere borrar la pantalla, entonces se ha de utilizar *milcd.clear()*. No tiene ni parámetros ni valor de retorno.

milcd.setCursor(): posiciona el cursor en la columna y fila especificadas como parámetros para escribir a partir de allí el próximo texto. Su primer parámetro es la columna en la que se quiere situar el cursor (la primera es la número 0) y su segundo parámetro es la fila (la primera es la número 0 también). Así pues, *milcd.setCursor(0,0)*; es equivalente a *milcd.home()*; No tiene valor de retorno.

milcd.cursor(): muestra el cursor en forma de línea de subrayado en la posición donde el próximo carácter será escrito. No tiene ni parámetros ni

EL MUNDO GENUINO-ARDUINO

valor de retorno. También existe la función **milcd.noCursor()**, la cual oculta el cursor (y tampoco tiene ni parámetros ni valor de retorno).

milcd.blink(): muestra el cursor parpadeando (suponiendo que este sea visible). No tiene ni parámetros ni valor de retorno. También existe la función **milcd.noBlink()**, la cual evita el parpadeo del cursor (y tampoco tiene ni parámetros ni valor de retorno).

milcd.display(): enciende la pantalla (después de haberla apagado mediante **milcd.noDisplay()**). Hay que tener presente que la función *milcd.noDisplay()* apaga la pantalla pero no pierde el texto actual (es decir, lo mantiene en la memoria interna de la pantalla –a diferencia de *milcd.clear()*–), por lo que si se ejecuta *milcd.display()* se volverá a restaurar el texto (y cursor) que se estaba mostrando previamente. Ninguna de las dos funciones tiene ni parámetros ni valor de retorno.

milcd.scrollDisplayLeft(): desplaza el contenido actual de la pantalla (texto y cursor) un espacio hacia la izquierda. No tiene ni parámetros ni valor de retorno. También existe la función **milcd.scrollDisplayRight()**, que desplaza el contenido actual un espacio hacia la derecha (y que tampoco tiene ni parámetros ni valor de retorno). Estas funciones son útiles cuando la longitud del texto a mostrar supera la anchura de la pantalla.

milcd.autoscroll(): activa el desplazamiento automático del contenido que se mostrará en pantalla a partir de este momento. Es decir, provoca que cada futuro carácter a mostrar sea "empujado" en un espacio por el siguiente, en el sentido marcado por las funciones *milcd.leftToRight()* o *milcdrightToLeft()* –ver párrafo siguiente–. El efecto visual es que cada carácter nuevo aparecerá en la misma posición dentro de la pantalla. No tiene parámetros ni valor de retorno. También existe la función **milcd.noAutoscroll()**, la cual evita este comportamiento (y tampoco tiene ni parámetros ni valor de retorno).

milcd.leftToRight(): establece el sentido de escritura del texto en la pantalla. Por defecto ya es de izquierda a derecha (es decir, que los caracteres que se vayan imprimiendo lo harán en ese sentido), por lo que no haría falta de entrada ejecutar esta función, a no ser que previamente se hubiera ejecutado **milcd.rightToLeft()**, la cual establece el sentido de escritura de derecha a izquierda. Ninguna de estas dos funciones afectará en cualquier caso al texto previamente escrito en la pantalla antes de su ejecución, y no tienen ni parámetros ni valor de retorno.

Veamos ahora varios casos de uso de algunas de las funciones anteriores, suponiendo las mismas conexiones entre placa Arduino y LCD. El siguiente código, por ejemplo, muestra un mensaje estático en la primera fila del LCD de forma permanente y un mensaje variable (en concreto, el número de segundos transcurridos desde el inicio de la ejecución del sketch) en la segunda fila del LCD:

Ejemplo 5.2

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
void setup() {
    milcd.begin(16,2);
    /*milcd.home();      Esta línea no es necesaria porque por defecto el cursor ya se sitúa allí.*/
    milcd.print("Hola");
}
void loop() {
    milcd.setCursor(0,1);      // Me situó en el primer carácter de la segunda fila
    milcd.print(millis()/1000); // Escribo los segundos desde el inicio del sketch
}
```

También podemos conseguir, por ejemplo, que un mensaje se visualice de forma parpadeante (en este caso, precediéndolo además con una cuenta atrás):

Ejemplo 5.3

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
byte cont;
void setup() {
    milcd.begin(16,2);
    for(cont=9; cont > 0; cont--) { //Cuenta atrás desde 9 hasta 0
        //Antes de imprimir el número 'i', me vuelvo a situar al final de la primera línea
        milcd.setCursor(15,0);
        milcd.print(cont);
        delay(1000);
    }
    milcd.print("Hola"); //Mensaje que haremos que parpadee (dentro de loop())
    delay(500);
}
void loop() {
    milcd.noDisplay();
    delay(500);
    milcd.display();
    delay(500);
}
```

EL MUNDO GENUINO-ARDUINO

El siguiente código muestra un mensaje moviéndose indefinidamente entre ambos extremos del LCD a modo de "rebote". En este caso concreto, los desplazamientos se realizan en periodos de medio segundo y hacen desaparecer el texto por completo (instante en el cual el movimiento cambia su sentido, volviendo entonces a aparecer el texto por pantalla otra vez):

Ejemplo 5.4

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
String frase="Hola";
byte numCols=16;
byte numFilas=2;
byte longFrage;
void setup() {
    milcd.begin(numCols,numFilas);
    milcd.print(frase);
    longFrage = frase.length();
}
void loop() {
    /*Realizo un movimiento de 4 posiciones (la longitud de la cadena mostrada) hacia la izquierda.
    De esta manera, la cadena desaparecerá por ese extremo */
    for (byte pos = 0; pos < longFrage; pos++) {
        milcd.scrollDisplayLeft();
        delay(500); //Dependiendo del tiempo indicado, el movimiento será más o menos rápido
    }
    /*Realizo un movimiento de 20 posiciones (longitud de la cadena (4) + longitud LCD (16))
    a la derecha. De esta manera, la cadena desaparecerá por completo tras recorrer todo el LCD */
    for (byte pos = 0; pos < longFrage + numCols; pos++) {
        milcd.scrollDisplayRight();
        delay(500);
    }
    /*Vuelvo a situar la cadena donde estaba inicialmente, desplazándola hacia la izquierda lo
    necesario (en este caso, la longitud del LCD (16)) */
    for (byte pos = 0; pos < numCols; pos++) {
        milcd.scrollDisplayLeft();
        delay(500);
    }
    //Espero un segundo antes de volver a empezar con el ciclo de movimientos
    delay(1000);
}
```

Otro ejemplo muy sencillo es el siguiente, donde el LCD muestra aquello que se escribe en el "Serial monitor" en tiempo real (suponiendo que nuestra placa Arduino la tenemos conectada a un computador vía USB):

Ejemplo 5.5

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
int luzFondo = 13; //Defino el pin para la luz de fondo
void setup() {
    pinMode(luzFondo, OUTPUT); digitalWrite(luzFondo, HIGH); //Activo la luz de fondo
    milcd.begin(16,2);
    Serial.begin(9600);
}
void loop() {
    if (Serial.available()>0) {
        delay(100);
        milcd.clear();
        while (Serial.available() > 0) {
            milcd.write(Serial.read());
        }
    }
}
```

El siguiente código muestra en el LCD una secuencia de caracteres que van posicionándose cada vez más a la derecha, a modo de barra de progreso animada:

Ejemplo 5.6

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
/*El código muestra en una posición tres caracteres personalizados, uno tras otro, y entonces paso a la siguiente posición y vuelvo a repetir la secuencia */
byte a[8] = {B00000,B00000,B00000,B00100,B00100,B00000,B00000,B00000};
byte b[8] = {B00000,B00000,B10001,B10001,B10001,B00000,B00000};
byte c[8] = {B11111,B10001,B10001,B10001,B10001,B10001,B10001,B11111};
void setup() {
    milcd.createChar(0,a);
    milcd.createChar(1,b);
    milcd.createChar(2,c);
    milcd.begin(16,2);
    milcd.clear();
}
void loop() {
    int i;
    for (int i=0; i<16; i++){
        milcd.setCursor(i,0); milcd.write(byte(0)); delay(250);
        milcd.setCursor(i,0); milcd.write(byte(1)); delay(250);
        milcd.setCursor(i,0); milcd.write(byte(2)); delay(250);
    }
}
```

```
    milcd.setCursor(i,0); milcd.write(byte(1)); delay(250);
    milcd.setCursor(i,0); milcd.write(byte(0)); delay(250);
}
delay(1000);
milcd.clear();
}
```

Librerías de terceros interesantes para usar con LCDs

Además de la librería "LiquidCrystal", existen varias librerías especializadas de terceros que pueden servirnos en un momento dado para definir de una forma más específica el comportamiento de nuestras pantallas LCD. Mencionaremos a continuación algunas de las más interesantes.

Para empezar, podemos comentar la existencia de la librería "New LiquidCrystal" (<https://bitbucket.org/fmalpartida/new-liquidcrystal>), la cual representa una mejora de la librería oficial de Arduino que permite, sobre todo, obtener una mayor velocidad de funcionamiento. También ofrece una mayor versatilidad al admitir otros protocolos de comunicación con LCDs diferentes de los modos 4-bit/8-bit (como, por ejemplo, los protocolos I²C o serie) con el consiguiente ahorro de cables.

NOTA: Para emplear dichos protocolos "alternativos", no obstante, además de la propia pantalla es necesario incluir en nuestro circuito un hardware específico –llamado genéricamente "backpack"– capaz de interpretarlos; esto lo estudiaremos en el próximo apartado.

La librería "LCDBitmap" (<https://bitbucket.org/teckel12/arduino-lcd-bitmap>), que puede funcionar tanto junto con la librería "LiquidCrystal" oficial como con la "New LiquidCrystal" mencionada en el párrafo anterior, añade la posibilidad de mostrar en las pantallas LCD de caracteres estándar imágenes formadas por píxeles individuales de luz, pudiéndose así visualizar líneas y figuras geométricas diversas (con relleno o sin él) de una forma muy flexible. Para ello, esta librería crea un array en la memoria SRAM de la placa Arduino para alojar todos los píxeles que formarán la imagen a mostrar (allí donde se especifique). Esta imagen está siempre formada (porque así lo dictamina esta librería) por un conjunto de 8 caracteres dispuestos en dos filas (4 caracteres en cada uno), por lo que, como las dimensiones de cada carácter son 5x8 píxeles, el tamaño total del array de píxeles en memoria (es decir, el de la imagen) es de (5x4)x(8x2)=20x16 píxeles.

Si lo que se quiere simplemente es mostrar caracteres en un tamaño mayor al estándar, se pueden usar las librerías "Phi_big_font" y "Phi_super_font", ambas

descargables de <http://code.google.com/p/phi-big-font>. La primera permite mostrar caracteres de un tamaño hasta 6 veces mayor y la segunda de hasta 40 veces mayor. La primera requiere el uso de dos filas y cuatro columnas para mostrar un carácter, con una columna en blanco para la separación entre uno y otro; esto significa que en una pantalla de 16x2 caben 4 caracteres y en una de 20x4 caben 5 en 2 filas. La segunda requiere el uso de cuatro filas y 5 columnas para mostrar un carácter, con una columna en blanco para la separación entre uno y otro; esto significa que en una pantalla de 20x4 caben 4 caracteres. Ambas librerías admiten animaciones. Otra librería similar (aunque útil solo para mostrar números grandes pero no letras) es <https://github.com/seanauff/BiNumbers>.

Merece destacar, finalmente, la existencia de varias librerías que permiten programar de forma muy sencilla elementos visuales en las pantallas LCD pensados para facilitar la interactividad con el usuario: menús de navegación, menús de selección de opciones, botones, cajas de texto, etc. Lógicamente, el método concreto de interactuación con estos elementos visuales puede ser muy variado (mediante el canal serie, mediante pulsadores, mediante sensores, etc.) pero, sea el que sea, estas librerías "gráficas" ofrecen un sistema de visualización coherente para el usuario. Ejemplos de ello son la librería "M2tklib" (<https://code.google.com/p/m2tklib>), la librería "Menwiz" (<https://github.com/brunialti/MENWIZ>), la "Arduino LCD Menu" (https://github.com/DavidAndrews/Arduino_LCD_Menu) o la "Arduino Menusystem" (<https://github.com/jonblack/arduino-menusystem>). Por otro lado, "LcdBarGraph" (<https://github.com/prampec/LcdBarGraph>) es una librería especializada en mostrar gráficas de barra.

Módulos LCD de tipo I²C o TTL-Serie

Las pantallas LCD comunes tienen el inconveniente de requerir muchos cables para conectarse al circuito. Como consecuencia, en nuestra placa Arduino nos pueden quedar pocos pines disponibles para usarlos en otras cosas. Una solución a este inconveniente es el empleo de pantallas LCD que utilizan un sistema de comunicación con la placa Arduino diferente del ya comentado, como pueden ser los protocolos I²C (el cual solo utiliza las líneas SDA y SCL), o el serie (usando solo las líneas RX y TX), principalmente.

En realidad, estas pantallas LCD son exactamente las mismas que las listadas en el apartado anterior, pero incorporan una pequeña plaquita en su dorso (llamada "backpack") que incluye un pequeño microcontrolador específicamente encargado de realizar la traducción del protocolo adecuado (I²C, Serie, etc.) a las señales "estándares" que la pantalla LCD entiende.

Backpacks I²C

Por ejemplo, DFRobot distribuye (con código de producto **DFR0063**) un conjunto formado por una pantalla alfanumérica de 2x16 caracteres más un backpack y otro conjunto (con código **DFR0154**) formado por una pantalla de 4x20 caracteres más el mismo backpack. Estos conjuntos pantalla+backpack tan solo requieren cuatro cables para ser controlados totalmente desde la placa Arduino (alimentación de 5V, tierra, cable SDA y cable SCL) y se programan mediante una librería propia (descargable de la página web de ambos productos) llamada "LiquidCrystal_I2C", prácticamente idéntica a la librería oficial de Arduino. Como dato interesante, destacar que el backpack dispone de un "trimpot" integrado para ajustar el contraste.

En realidad, el backpack incorporado en estos dos productos anteriores también puede ser adquirido individualmente sin la LCD adosada como producto de DFRobot con código **DFR0175**. Esto nos puede venir muy bien si ya poseyéramos previamente alguna pantalla compatible con el chip Hitachi HD44780 y quisieramos conectarla a nuestro circuito usando el menor número de cables posibles: simplemente acoplando dicho backpack a los pines (*rs*, *re*, *en*,...) de la pantalla en cuestión conseguiríamos ese objetivo. Un backpack similar al anterior (de hecho, está basado en el mismo chip: el PCF8574 del fabricante NXP, aunque tiene el interesante añadido de incorporar un "jumper" para poder activar/desactivar fácilmente la luz de fondo) es el que podemos adquirir en Tronixlabs como producto **TLLCDI2C** (o también en MinilntheBox pero allí como producto **#00643045**). Incluso el propio chip PCF8574, al comercializarse en formato DIP –en Tronixlabs, por ejemplo–, podría ser utilizado directamente en circuitos sobre una breadboard sin ni siquiera tener que utilizar su backpack asociado (aunque en este caso deberíamos consultar su "datasheet" para conocer la funcionalidad de cada una de sus patillas y realizar las conexiones pertinentes) y podría ser programado mediante la misma librería "LiquidCrystal_I2C" (o también la "New LiquidCrystal" anteriormente mencionada).

Otro conjunto de pantalla+backpack es el "**TWILCD**" de Akafugu; este producto también se puede conectar a nuestra placa Arduino mediante solo cuatro cables (alimentación, tierra, SDA y SCL) y también se puede programar mediante una librería propia muy parecida a la librería oficial de Arduino (descargable en este caso de <https://github.com/akafugu/twilcd>). La diferencia más relevante con el producto de DFRobot está en que el backpack del "TWILCD" carece de potenciómetro manual para regular el contraste y tampoco ofrece ningún "jumper" para activar/desactivar la luz de fondo pero, a cambio, permite controlar ambas características desde el software. Al igual que DFRobot, Akafugu nos permite tanto adquirir el backpack junto con la pantalla alfanumérica como también ambos componentes por separado; de todas formas, hay que tener en cuenta que el backpack ofrecido solo es compatible

con LCDs Hitachi HD44780 (o similares) cuyas dimensiones sean 1x16 o 2x16 o 1x20 o 2x20; si queremos manejar LCDs de tipo RGB o bien de un tamaño mayor (como 4x40, por ejemplo), Akafugu ofrece otro backpack (también con pantalla opcional) llamado "**TWILCD 40x2/40x4/RGB**".

Adafruit distribuye un backpack de tipo I²C por separado (producto nº 292, basado en el chip MCP23008), pero no ofrece la posibilidad de adquirirlo junto con una pantalla ya preensamblada, sino que es necesario soldarlo manualmente con alguna de las pantallas LCD alfanuméricas existentes en su catálogo. En todo caso, para poderlo controlar es necesario utilizar una librería propia (muy similar a la oficial de Arduino) descargable de <http://github.com/adafruit/LiquidCrystal>.

Backpacks serie

Por otro lado, también existen pantallas LCD que, gracias a un chip TTL-UART que llevan incorporado, implementan el protocolo serie. De esta manera, con solamente tres cables (alimentación, tierra y la conexión del pin RX de la pantalla al pin TX de la placa Arduino) ya podemos controlarlas. Tan solo es cuestión de enviarles mediante *Serial.write()* algunos comandos propios específicos para cada acción que deseemos realizar (activar luz de fondo, limpiar pantalla, posicionar el cursor, etc.), y mediante *Serial.print()* el texto a imprimir. Para conocer estos comandos se ha de consultar el datasheet respectivo. Sparkfun, por ejemplo, distribuye varios modelos de este tipo, como los productos nº 9393, nº 9394, nº 9395, nº 9396 o nº 9568, entre otros. El siguiente código es un ejemplo de prueba para cualquiera de estas pantallas:

Ejemplo 5.7

```
/*El texto a mostrar se envía por el canal serie directamente. Los comandos de control se especifican con dos bytes: el primero es un comando genérico que "advierte" al LCD del tipo de acción a realizar, y el segundo es el comando que realiza esa acción. Es conveniente utilizar Serial.write() debido a que los comandos son en realidad valores hexadecimales (con Serial.print() se podrían confundir con su representación ASCII). Los delay() son necesarios porque si los comandos se envían demasiado seguidos, el LCD puede no recibirlos correctamente.*/
void setup(){
    Serial.begin(9600); //El LCD funciona por defecto a esa velocidad
    luzfondoOn(); irLineaUno(); Serial.print("Hola");
    irLineaDos(); Serial.print("Bienvenido"); delay(100);
}
void loop(){}
//Enciende la luz de fondo
void luzfondoOn(){
    //Comando de control para gestionar la luz de fondo
    Serial.write(0x7C);
```

EL MUNDO GENUINO-ARDUINO

```
//Comando que establece la máxima cantidad de luz posible
Serial.write(157); delay(5);
}

//Apaga la luz de fondo
void luzfondoOff(){
    Serial.write(0x7C);
    Serial.write(128); //128 es el valor mínimo (luz apagada)
    delay(5);
}

//Coloca el cursor en el carácter 0 de la primera línea
void irLineaUno(){
    //Comando de control para gestionar la posición del cursor
    Serial.write(0xFE);
    //Comando que cambia efectivamente la posición a la indicada
    Serial.write(128); delay(5);
}

//Coloca el cursor en el carácter 0 de la segunda línea
void irLineaDos(){
    Serial.write(0xFE); Serial.write(192); delay(5);
}

//Coloca el cursor en cualquier posición (la primera fila es la nº 0 y la primera columna también)
void irPosicion(int fila, int col) {
    //Comando de control para gestionar la posición del cursor
    Serial.write(0xFE);
    Serial.write((col + fila*64 + 128)); //Posición deseada
    delay(5);
}

//Otra función interesante. Para saber más, consultar el datasheet
void borrarLCD(){
    //Comando de control para gestionar el borrado del LCD
    Serial.write(0xFE);
    //El comando propiamente dicho que borra la pantalla
    Serial.write(0x01); delay(5);
}
```

De todas formas, existe una librería llamada "SerLCD" que permite utilizar los LCDs serie de Sparkfun mediante (casi) las mismas funciones que las de la librería "LiquidCrystal" oficial de Arduino, facilitando enormemente su gestión. Se puede descargar de aquí: <https://github.com/nemith/serLCD>.

Otro backpack serie (con pantalla LCD opcionalmente ya preensamblada) es el "**LCD117 Serial LCD Kit**" de Modern Device; en la página del producto se ofrece la referencia completa de comandos que acepta vía *Serial.write()*, junto con varios

códigos Arduino de ejemplo y, además, un software específico para el diseño de caracteres personalizados. Otros productos que pueden recibir los comandos de control y texto a través de *Serial.write()*—y que tienen la pantalla LCD ya preensamblada de fábrica— son los nº 27976, nº 27977 y nº 27979 de Parallax.

Otro ejemplo de pantalla LCD más backpack serie (en este caso, en forma de kit) controlable mediante *Serial.write()* es el producto nº 782 (con letras negras sobre fondo RGB) o el producto nº 784 (con letras RGB sobre fondo negro) de Adafruit. Ambos productos tienen, no obstante, la particularidad de disponer, además del backpack serie, de un backpack USB. De esta manera, el LCD se puede comunicar (a través de un cable USB estándar) directamente con un computador para recibir de él (gracias a algún software de tipo "terminal serie") los comandos de control o texto.

En el caso de querer adquirir tan solo el backpack serie/USB (porque ya dispongamos de la pantalla LCD), podríamos hacernos, por ejemplo, con el producto **LCD72858P** de Seeedstudio.

Finalmente, otro módulo LCD digno de destacar es el "**3-Wire Serial LCD module**" de DFRobot, que, tal como su nombre indica, también solo requiere el uso de tres cables. Incorpora una pantalla gráfica de 128x64 píxeles y es programable mediante una librería particular suya (derivada de la "SPI" oficial), descargable de la web del producto.

Shields que incorporan LCDs

A parte de los módulos LCD ya mencionados o similares, podemos adquirir para nuestros proyectos shields que integran una LCD y que tienen la ventaja de ser directamente empotrables sobre nuestra placa Arduino. De esta manera, tendremos un circuito más compacto. Además, estos shields no incorporan solo una pantalla sino que además suelen venir con un conjunto de pulsadores (como mínimo cuatro para hacer la función de cursores por la pantalla y poder navegar por los menús, además de otro botón más de control, para seleccionar la opción elegida) que permiten añadir interactividad a nuestros proyectos y poder así enviar órdenes a la placa Arduino sin necesitar ningún computador externo. A continuación, se nombran algunas de las diferentes alternativas que tenemos.

DFRobot fabrica su "**LCD keypad shield**", que incorpora una pantalla LCD alfanumérica de 2x16, de fondo azul y caracteres blancos, con soporte para ajuste de contraste mediante potenciómetro y activación o desactivación de la luz de fondo (conviene saber que DFRobot también fabrica el llamado "**PCB of LCD Keypad**

EL MUNDO GENUINO-ARDUINO

shield", que consiste en el mismo shield pero sin pantalla LCD incorporada, para tener así la libertad de quitar/poner eventualmente otras posibles pantallas compatibles). Este shield está controlado por el chip HD44780 en modo "4-bit", por lo que la pantalla existente se puede programar mediante la librería "LiquidCrystal" oficial. En este sentido, como los pines del shield del 4 al 10 son usados para comunicarse internamente con la pantalla, no se pueden aprovechar para otra cosa, pero para compensar esta pérdida de pines, este shield aporta a cambio varios pines analógicos de entrada extra. Por otro lado, este shield tiene además 5 con botones de control (arriba, abajo, derecha e izquierda y selección) y un botón de reinicio; para saber qué botones se están pulsando en un momento dado, nuestro programa debe ir leyendo en cada "loop" la señal recibida por el pin analógico nº 0: según el valor numérico (en diferentes rangos de 0 a 1023) que tenga esa señal, se podrá saber qué botón ha sido pulsado y actuar en consecuencia.

Otros shields prácticamente idénticos al anterior (ya que utilizan el mismo chip HD44780 en modo "4-bit" –y, por tanto, son igualmente programables mediante la librería oficial de Arduino–, también tienen pantallas de 2x16 e incorporan un potenciómetro y botones de dirección y control gestionables a través de una entrada analógica) son el "**1602 LCD shield**" de IteadStudio, el "**16x2 LCD Keypad Shield B**" de LinkSprite, el "**LCD Keypad shield**" de GorillaBuilderz o el "**LCD & Keypad shield**" de Freetronics. Para gestionar los pulsadores, estos shields utilizan la misma técnica (mencionada en el párrafo anterior) de leer en cada "loop" la señal recibida por el pin analógico nº 0, comprobar su valor numérico y actuar en consecuencia (en las respectivas páginas del producto se ofrecen códigos de ejemplo mostrando esta técnica). En el caso particular del shield de Freetronics, no obstante, existe la posibilidad de utilizar, en vez de la librería oficial, otra librería específica (<http://rweather.github.com/arduinolibs/classLCD.html>) que, entre otras ventajas, permite controlar las pulsaciones de los botones de una forma más sencilla simplemente invocando a la función *milcd.getButton()* y procesando su valor devuelto (que representa el identificador del botón pulsado).

Los "**RGB LCD shield kit**" de Adafruit (productos nº 714, nº 716 y nº 772) son aparentemente muy parecidos a los shields anteriormente mencionados (incorporan el chip HD44780 o compatible, una pantalla 2x16, un potenciómetro para regular el contraste, 5 botones cuyas pulsaciones son leídas por una sola entrada analógica, etc.) pero aportan dos novedades: una es que permiten que su pantalla pueda cambiar su color de fondo (algo que en los otros shields no es posible) y la otra –más importante– es que la comunicación que establecen con la placa Arduino es mediante el protocolo I²C en vez de mediante el modo "4-bit". Esto es así porque estos shields incorporan el backpack I²C de Adafruit mencionado en el apartado anterior (su producto nº 292, recordemos, basado en el chip MCP23008). Gracias a la presencia

de este backpack, estos shields de Adafruit tan solo necesitan dos pines para comunicarse con la placa Arduino acoplada debajo (concretamente, en el caso de la placa UNO, los pines analógicos nº 4 y nº 5), por lo que dejan así más pines libres para otros usos. No obstante, el hecho de que la comunicación con la placa se realice vía I²C obliga a utilizar una librería propia diferente de la librería "LiquidCrystal" oficial, la cual se puede descargar desde aquí: <https://github.com/adafruit/Adafruit-RGB-LCD-Shield-Library>; esta librería es idéntica funcionalmente a la librería oficial, pero ofrece un par de funciones extra interesantes dignas de mención: *milcd.setBacklight(...)* –para establecer el color de fondo– y *milcd.readButtons()*–para reconocer el botón pulsado de una forma cómoda (a partir de su valor devuelto, que representa el identificador de dicho botón) –.

Otro shield que utiliza el mismo chip MCP23008 que los shields de Adafruit (y que, por tanto, puede ser programado mediante su misma librería) es el "**I2C LCD Shield**" de MakerStudio, el cual permite como novedad el acople/desacople de la pantalla LCD (pudiendo así sustituir esta sin necesidad de cambiar el shield completo). Otro shield muy similar (que sustituye los cinco botones por un joystick de 5 posiciones) es el "**DeuLigne**" de Snootlab, el cual puede programarse con la librería propia descargable de <https://github.com/Snootlab/Deuligne>, la cual permite, además de gestionar la pantalla, manejar de una forma integral las pulsaciones de los botones.

Shields y módulos que incorporan GLCDs

DFRobot ofrece su "**LCD12864 Shield**", que incorpora una pantalla GLCD de 128x64 píxeles (basada en el chip ST7565) y un joystick de 5 posiciones (controlado a través de un único pin de entrada analógica, el nº 0). Este shield se alimenta a través del pin 3.3V de la placa Arduino y se comunica con esta mediante SPI (a través de los pines digitales nº 9, nº 10, nº 11 y nº 13), monopolizando además los pines nº 7 (para activar/desactivar la luz de fondo) y nº 8 (para un eventual reseteo del shield); no obstante, a cambio aporta 5 pines de entrada analógica y 8 pines digitales extra. Para programar este shield se usa la librería "U8glib" (<https://code.google.com/p/u8glib>), la cual incluye un variado conjunto de funciones que permiten dibujar círculos, rectángulos, líneas y cadenas de caracteres de diferentes colores y tamaños de una forma muy sencilla para distintos modelos de GLCDs (no solo la de este shield).

IteadStudio por su parte distribuye los shields llamados "**IBridge Lite**" e "**IBridge**", ambos diseñados para ser empleados junto con la pantalla "**Nokia 5110 LCD**" (comercializada también por IteadStudio por separado) pero con diferente número de botones (en el primer caso, estos están dispuestos ordenadamente en 3

EL MUNDO GENUINO-ARDUINO

filas de 3 botones cada una – $3 \times 3 = 9$ – y en el segundo caso, en 4 filas de 4 botones – $4 \times 4 = 16$ –). Tanto los gráficos mostrados en la pantalla de los dos shields como el reconocimiento de las pulsaciones de sus respectivos botones se pueden controlar mediante una librería básica descargable de la página web de cada producto bajo en nombre de "Demo code".

Si preferimos, en cambio, utilizar módulos independientes en vez de shields, la pantalla "Nokia 5110 LCD" de IteadStudio mencionada en el párrafo anterior podría servirnos perfectamente por sí sola. Esta misma pantalla (de la cual destacaremos su tamaño de 84x48 píxeles y el estar basada en el chip PCD8544 de NXP) también está distribuida por Sparkfun (con código nº 10168). Concretamente, las conexiones necesarias para poderla controlar desde una placa Arduino UNO son las siguientes:

Pantalla	Placa Arduino UNO
VCC (nº 1)	3.3V
GND (nº 2)	GND
SCE (nº 3)	Un pin digital ⁽¹⁾
RST (nº 4)	Un pin digital ⁽²⁾
D/C (nº 5)	Un pin digital ⁽²⁾
MOSI (nº 6)	Pin digital nº 11 ⁽²⁾
SCLK (nº 7)	Pin digital nº 13 ⁽²⁾
LED (nº 8)	Un pin PWM ⁽³⁾ o 3.3V

⁽¹⁾ A través de una resistencia en serie de 1KΩ

⁽²⁾ A través de una resistencia en serie de 10KΩ

⁽³⁾ A través de una resistencia en serie de 330Ω

El pin "VCC" de la pantalla anterior alimenta los circuitos internos de la LCD (los cuales han de estar sometidos a una tensión entre 2,7V y 3,3V y consumen entre 6 y 7mA) y el pin "LED" alimenta, por su parte, a los cuatro LEDs que iluminan el fondo de la pantalla (los cuales han de estar sometidos a la misma tensión entre 2,7V y 3,3V –de ahí la necesidad de la resistencia en serie si se conectan a un pin PWM que controle su iluminación– y pueden llegar a consumir en total hasta 80mA). El sistema de comunicación que implementa no es exactamente SPI pero se parece: existe una línea de reloj (SCLK), una de datos (MOSI) y una de tipo "Slave-Select" (SCE) funcionando en modo "activo BAJO" –al igual que lo hace la línea de reseteo (RST)–; todas estas líneas funcionan a 3,3V, de ahí la necesidad de la resistencia en serie. Además, existe otra línea de entrada (D/C) cuya tarea es indicar a la pantalla si los datos que está recibiendo por la línea MOSI son comandos de control (línea D/C en estado BAJO) o mensajes a mostrar (línea D/C en estado ALTO). El conjunto de comandos de control es variado (borrar pantalla, apagarla, invertir los píxeles, etc.) y

para conocerlos deberíamos consultar su datasheet, pero, afortunadamente, disponemos de varias librerías que nos evitan este trabajo, como por ejemplo el código de muestra https://github.com/sparkfun/GraphicLCD_Nokia_5110, la librería <https://github.com/carlosefr/pcd8544> o la librería de Adafruit (que, aunque está diseñada especialmente para la pantalla distribuida por ellos mismos –con código de producto nº 338–, es la librería más completa y versátil de todas al permitir dibujar líneas y figuras geométricas) <https://github.com/adafruit/Adafruit-PCD8544-Nokia-5110-LCD-library>, la cual, eso sí, necesita para funcionar que esté instalada también la librería base "Adafruit GFX", ya conocida.

El problema de conectar un módulo cuya circuitería interna solo soporta hasta 3,3V (como es el caso de la pantalla Nokia) a una placa Arduino cuyos pines funcionan en niveles de tensión de hasta 5V es que fácilmente podemos "despistarnos" y cometer el error de no regular convenientemente esta diferencia de tensiones admitidas y, por tanto, de quemar el módulo en cuestión. Por eso, en la tabla y el párrafo anteriores se indica la necesidad de interponer entre la pantalla y la placa Arduino una resistencia por cada línea de conexión: para que así la tensión de la señal recibida por esa pantalla se reduzca previamente. No obstante, este "apaño" solo nos será útil en circuitos donde las señales se transmitan por líneas de conexión funcionando un solo sentido, de tensión mayor a tensión menor. En el caso de la pantalla Nokia, eso es así (ya que es la placa Arduino la que envía todas las señales hacia la pantalla), pero si hubiera habido alguna línea de conexión que transmitiera una señal en sentido contrario (es decir, de la pantalla hacia la placa Arduino), tendríamos que poder adaptar la señal original (de una tensión menor) a una señal de tensión mayor, algo que una resistencia no puede hacer. En esta situación (por otra parte, muy habitual al querer comunicar dispositivos vía TTL-Serie, SPI o I²C, por ejemplo) no quedaría más remedio que recurrir a otro componente electrónico llamado "convertidor de nivel bidireccional".

Breve nota sobre los convertidores de nivel bidireccionales

Un chip convertidor de nivel bidireccional (en inglés, "bi-directional level shifter") tiene la función de regular la tensión entre dos componentes electrónicos cuyo voltaje de trabajo es diferente. Un caso típico es el que acabamos de comentar: conectar la placa Arduino (cuyos pines de entrada/salida funcionan todos con una señal de 5V) con un módulo externo (como puede ser la pantalla GLCD mencionada anteriormente) cuya alimentación y pines de entrada/salida funcionan a 3,3V. Tal como ya se ha comentado, si se conectaran los pines de la placa Arduino directamente a los pines de ese módulo, estos recibirían más tensión de la soportada y se quemarían. Y en dirección contraria también hay

problemas: si la placa Arduino recibiera directamente señales provenientes del módulo, en ocasiones un nivel HIGH algo por debajo de los 3,3V podría confundirse con un nivel LOW en el rango de 5V. Por tanto, un convertidor de nivel bidireccional permite "empalmar" varios pines funcionando a 5V a varios pines funcionando a 3,3V (y viceversa, de ahí el nombre de "bidireccional") sin que aparezcan estos problemas.

Los convertidores de nivel bidireccionales son chips que pueden estar disponibles en el mercado en diversas formas. Lo más habitual, no obstante, es que aparezcan integrados dentro de una plaquita breakout para su mejor manipulación. Todas ellas disponen de un conjunto de patillas (normalmente entre cuatro y ocho) donde podremos conectar los pines del componente que trabaja a una tensión mayor, otro conjunto de patillas donde podremos conectar los pines del componente que trabaja a una tensión menor, un par de patillas donde se recibe las dos alimentaciones y varias patillas para tierra. Por ejemplo, Adafruit distribuye una plaquita breakout como producto **nº 395** (basado en el chip TXB0108, el cual dispone de hasta 8 líneas bidireccionales) y otra como producto **nº 1875** (basado en el chip TXB0104, que tiene 4 líneas bidireccionales). Sparkfun por su lado, distribuye una plaquita con código de producto **nº 11771** (basado también en el chip TXB0104). Todos ellos funcionan perfectamente para comunicar componentes trabajando a 5V con otros trabajando a 3,3V (y otros voltajes, dependiendo del modelo).

En el caso particular de querer conectar canales I²C, de manera que, por ejemplo, podamos empalmar un dispositivo I²C funcionando a 3,3V a los pines SDA y SCL de la placa Arduino (que funcionan a 5V), deberíamos utilizar unos convertidores de nivel específicos para I²C, ya que este protocolo necesita un sistema interno de resistencias "pull-up" que en otro tipo de comunicaciones como la TTL-Serie (líneas RX,TX) o la SPI (líneas CS, MOSI, MISO y SCK) no es necesario. Ejemplos de estos convertidores específicos son el producto **nº 757** de Adafruit, el producto **nº 12009** de Sparkfun o el "**Logic level converter module**" de Freetronics (todos basados en cuatro transistores BSS138 que proporcionan cuatro líneas independientes), y además específicamente el producto **nº 11955** de Sparkfun (basado en el chip PCA9306) o el **I2C-TRN-V2** de Gravitech (basado en el chip PCA9512)

NOTA: Hay que tener en cuenta varias consideraciones:

*Los convertidores de nivel tan solo funcionan para señales digitales, no convierten señales analógicas. Para lograr esto, se ha de usar un componente diferente: un amplificador operacional (comúnmente llamados "op-amp").

*Existen convertidores de nivel que no son bidireccionales (esto es, que funcionan en solo un sentido -concretamente desde la tensión mayor a la menor-, de forma similar a como lo haría -por cada una de las líneas- un divisor de tensión). Un ejemplo de ello son los chips (todos distribuidos en formato DIP) 75LVC245 (producto nº 735 de Adafruit) o 74HC4050 de Texas Instruments, o el chip HEX4050BP de NXP (estos últimos disponibles en Mouser, Jameco o distribuidores similares).

*Existen determinados componentes cuya tensión de alimentación es de 3,3V pero no necesitan convertidores de nivel para funcionar, debido a que sus pines de entrada/salida son compatibles con 5V. En esos casos, lo único que debemos procurar es alimentar la placa a la tensión adecuada y nada más. Pero hay que asegurarse primero de que, efectivamente, los pines de entrada/salida admitan tensiones de 5V.

Breve nota sobre la visualización de imágenes "al vuelo"

Un problema que tiene la pantalla Nokia 5110 estudiada en los párrafos anteriores es que carece de zócalo microSD (o de cualquier otro tipo de sistema de almacenamiento) para alojar ficheros de imagen. Esto obliga a realizar los siguientes pasos para poder visualizar en ella fotografías (las cuales, aun así, solo podrán ser de 2 "colores": el color "píxel iluminado" y el color "píxel apagado"):

1. Convertir la imagen deseada (con nuestro editor de imágenes preferido) al formato BMP monocromo (también conocido como "BMP blanco/negro" o "BMP 1-bit" –porque solo se emplea 1 bit para definir el color de cada píxel–). Si la imagen estuviera en otro formato diferente (JPG, PNG, etc.) los pasos siguientes no se podrían realizar con éxito porque internamente son demasiado complejos: el formato BMP es un formato muy simple y sin compresión, ideal para que las placas Arduino puedan manipularlo sin problemas.

NOTA: Gimp (<http://www.gimp.org>) es un editor de imágenes libre y multiplataforma que nos puede servir para realizar este punto. Concretamente, si abrimos la fotografía en cuestión con este programa, para convertirla al formato adecuado debemos ir al menú "Image->Mode->Indexed" y en el cuadro que aparece seleccionar "Use black and white (1-bit) palette". Seguidamente, debemos ir al menú "File->Export as" y grabar la imagen como fichero BMP.

2. Redimensionar esta imagen a un tamaño igual o menor que la resolución de la pantalla (en este caso, 84x48).

3. Transformar los bits de la imagen BMP en un array de $(84 \times 48)/8 = 504$ códigos numéricos. Para ello deberemos usar un programa –solo para Windows– llamado "LCD Assistant" (disponible en http://en.radzio.dxp.pl/bitmap_converter) u otro programa –multiplataforma– llamado "Img2Code" (descargable desde <https://github.com/ehubin/Adafruit-GFX-Library/tree/master/Img2Code>) o bien

la aplicación online <http://manytools.org/hacker-tools/image-to-byte-array>. En los tres casos, tras cargar la imagen deseada, la aplicación nos dará a elegir su orientación final (horizontal o vertical) pero si en concreto usamos el "LCD Assistant", además deberemos indicar el tamaño final deseado de la imagen (84x48 o menor) y asegurarnos de que la "size endianess" valga "Little" (el resto de valores de configuración que aparecen –como por ejemplo el número de píxeles definidos en cada byte, cuyo valor por defecto es 8– se pueden dejar inalterados). Una vez realicemos la conversión, en los tres casos obtendremos un fichero de texto cuyo contenido es la declaración de un array (que debería ser de tipo *char* solamente, ni *unsigned* ni *const*) inicializado con esos 504 valores.

4. Insertar la declaración/inicialización de ese array dentro de la sección de declaraciones de variables globales de nuestro sketch y escribir (con los parámetros adecuados y allí donde sea necesario dentro de ese mismo sketch) la función *milcd.drawString()*, que pertenece a la librería "Adafruit GFX" y que se encarga de generar, a partir del array indicado, la imagen correspondiente y de mostrarla por pantalla de forma automática.

Otra pantalla GLCD que viene en forma de módulo es el producto nº **710** de Sparkfun, la cual tiene unas dimensiones de 128x64 e incorpora el chip controlador KS0108 de Samsung. Este módulo posee 20 pines de conexión, los cuales se han de conectar (del nº 1 al nº 20) a los siguientes pines-hembra de la placa Arduino: 5V, GND, patilla central de un potenciómetro de 10KΩ regulador de contraste, D8, D9, D10, D11, D4, D5, D6, D7, A0, A1, RST, A2, A3, A4, una patilla lateral del potenciómetro (la otra va a tierra), 5V y GND. Para programar sketches que hagan uso de este módulo, la librería "Openglcd" (<https://bitbucket.org/bperrybap/openglcd>) así como también la ya mencionada "U8glib" son excelentes opciones: con ambas podremos dibujar rectángulos, círculos, líneas, puntos... y cadenas de caracteres de múltiples colores y tamaños. Estas dos librerías (que, de hecho, son compatibles con cualquier otro módulo que incorpore el chip KS0108 y más) están muy bien documentadas en sus respectivas páginas oficiales, por lo que remito al lector a su consulta.

Por otro lado, Sparkfun también ofrece su producto nº **8799** (GLCD monocromática de 160x128, gestionada por el chip T6963C). Lo más interesante de este producto es que acoplándole un módulo llamado "Graphic Serial LCD Back Pack" (producto nº **9352**) se puede establecer comunicación en serie con ellas. Esto quiere decir que tan solo necesitaremos (además del cable de alimentación y de tierra) un cable que conecte el pin TX de nuestra placa Arduino con el pin RX del backpack para poderles enviar mediante *Serial.write()* comandos hexadecimales que controlen la

visualización (como por ejemplo el color de fondo y de primer plano, la cantidad de brillo, el borrado de la pantalla, etc.) y que impriman los textos y los dibujos deseados (líneas, círculos, rectángulos, etc.) en la ubicación pertinente. Para conocer cuáles son estos comandos hexadecimales deberemos consultar el datasheet del producto, pero, afortunadamente tenemos la opción, para este modelo de pantalla, de utilizar una sencilla librería que evitará tener que escribirlos directamente: la librería "Serialglcdlib" (<http://sourceforge.net/projects/serialglcdlib>).

Sparkfun distribuye también su producto nº **8884**, que no es más que la pantalla GLCD mencionada en el párrafo anterior junto con el mismo backpack serie mencionado también en el párrafo anterior, todo en uno de forma integrada. Otra pantalla GLCD (más pequeña, de 128x64) que también incorpora el mismo backpack de fábrica (y que, por tanto, igualmente solo requiere tres cables para funcionar: alimentación, tierra y datos serie) es el producto nº **9351** de Sparkfun.

USO DE PANTALLAS TFT

Las pantallas TFT (del inglés, "Thin Film Transistor") son un tipo concreto de pantallas de cristal líquido (es decir, de LCDs; por eso, a menudo se mencionan con el acrónimo completo "TFT-LCD") cuya particularidad principal es que internamente utilizan uno o más transistores para controlar cada píxel, mejorando así la calidad de la imagen mostrada (en aspectos como el contraste, la frecuencia de refresco o la resolución) respecto a las LCDs de caracteres o GLCDs vistas en el apartado anterior. Más específicamente, las pantallas TFT son LCDs que incorporan una tecnología de tipo "matriz activa" en contraste con las otras, que es de tipo "matriz pasiva". Su mayor inconveniente es su (mayor) precio.

Shields y módulos que incorporan pantallas TFT

Adafruit distribuye su "1.8" TFT Shield" con producto nº **802**. Este shield incorpora un zócalo para insertar una tarjeta microSD (que ha de estar formateada en FAT16 o FAT32 para poder guardar en ella imágenes de tipo BMP 24 bits) y una pantalla TFT de 128x160 píxeles y 1,8 pulgadas de diagonal que permite mostrar 2^{18} colores diferentes. Además, incorpora un joystick de cinco posiciones (arriba, abajo, derecha, izquierda, selección). El controlador que lleva (el ST7735 de Sitronix) trabaja a 3,3V, pero el shield incorpora un convertidor de nivel que permite no preocuparnos de este tema. Por tanto, para usar este shield tan solo tenemos que encavarlo sobre la placa Arduino y listo. Como solo se apropia de 4 pines digitales de nuestra placa Arduino para la transmisión de datos vía SPI (los nº 8 –"D/C"–, nº 10 –"TFT CS"–,

EL MUNDO GENUINO-ARDUINO

nº 11 –"MOSI"– y nº 13 –"SCK"–), del pin analógico nº 3 para el control del joystick y de los pines digitales nº 4 y nº 12 para la gestión de la tarjeta microSD (líneas "SD CS" y "MISO", respectivamente), deja aún mucho margen para poder seguir utilizando bastantes entradas y salidas de Arduino.

Para poder trabajar con este shield necesitamos dos librerías: la librería "Adafruit ST7735" (encargada de la gestión interna de la comunicación hardware con el chip ST7735) y la librería "Adafruit GFX" (responsable del dibujado –a todo color– de diferentes formas geométricas –concretamente, de píxeles, líneas, rectángulos, triángulos y círculos–, así como de la visualización de ficheros de imagen y de la impresión de textos, todo ello en varios tamaños y orientaciones y –en el caso de los textos– diferentes fuentes de letra posibles).

Breve nota sobre la librería "Adafruit GFX"

La librería genérica "Adafruit GFX" siempre necesita para funcionar otra librería complementaria (también desarrollada por Adafruit) que es específica del modelo concreto de pantalla utilizado (GLCDs, OLEDs, TFTs, etc.; la lista completa está en <http://learn.adafruit.com/adafruit-gfx-graphics-library>). Esta separación entre una única librería gráfica común y diferentes librerías específicas (a elegir una para un determinado hardware) permite disponer de una sola sintaxis de programación sea cual sea el tipo de pantalla, ya que el desarrollador tan solo necesitará escribir las funciones ofrecidas por "Adafruit GFX" sin tener que preocuparse por los detalles técnicos internos del hardware utilizado, que quedan ocultos tras ella. De esta manera, los sketches Arduino se pueden adaptar fácilmente con los mínimos cambios. Las funciones de la librería "Adafruit GFX" son bastante intuitivas (*milcd.drawLine()*, *milcd.drawCircle()* o *milcd.fillTriangle()*, por ejemplo) pero si el lector quiere profundizar en su estudio, recomiendo consultar el enlace anterior.

Otra librería que también ofrece funciones de impresión de texto y de dibujado de figuras (líneas, polígonos, círculos, etc.) para multitud de pantallas diferentes pero que, además, incluye funciones específicas para diseñar elementos gráficos interactivos (como botones, etiquetas, menús, barras de desplazamiento, etc.) es la llamada "LCD_Screen" (http://embeddedcomputing.weebly.com/lcd_screen-download.html)

Si se desea, se puede utilizar un módulo independiente (una "placa breakout") con la misma pantalla, chip controlador, zócalo de tarjeta microSD y circuitería reguladora de voltaje que el shield anterior, pero sin el joystick. Se

CAPÍTULO 5: LIBRERÍAS ARDUINO

programa además con las mismas librerías "Adafruit ST7735"/"Adafruit GFX". Es el producto nº 358 de Adafruit. Este módulo se comunica (también) vía SPI con la placa; en este caso solo hay que tener la precaución de conectar bien los cables de la siguiente manera:

Placa breakout	Placa Arduino UNO
LITE	Un pin PWM o 5V (para luz de fondo)
MISO	Pin digital nº 12 (si se usa tarjeta SD)
SCK	Pin digital nº 13
MOSI	Pin digital nº 11
TFT_CS	Un pin digital (a definir en el código)
CARD_CS	Un pin digital (a definir en el código, si se usa tarjeta SD)
D/C	Un pin digital (a definir en el código)
RESET	Un pin digital (a definir en el código)
VCC	5V
GND	GND

Otro módulo basado también en el chip ST7735 (y por tanto, programable mediante las mismas librerías "Adafruit ST7735"/"Adafruit GFX" o "TFT" oficial) es el producto nº 2088 de Adafruit. Esta plaquita breakout (que trabaja a 3,3V pero que incorpora un convertidor de nivel para no preocuparnos de este detalle) también incluye, al igual que los módulos anteriores, un zócalo para insertar una tarjeta microSD (que ha de estar formateada en FAT16 o FAT32 para poder almacenar imágenes –de tipo BMP 24 bits– en su interior) pero en este caso su pantalla TFT es de 128x128 píxeles, 1,4 pulgadas y capaz de mostrar solamente 2^{16} colores. Las conexiones son idénticas a las mostradas en la tabla anterior (de hecho, hasta la nomenclatura de cada pin del módulo es igual, excepto "VCC" que pasa a ser "VIN").

Otro módulo también distribuido por Adafruit con pantalla TFT integrada (en este caso de 320x240 píxeles, 2,2 pulgadas de diagonal y capaz de mostrar 2^{18} colores diferentes) es su producto nº 1480. Esta plaquita breakout incorpora, al igual que los productos anteriores, un zócalo para insertar una tarjeta microSD (que igualmente ha de estar formateada en FAT16 o FAT32 para poder almacenar imágenes –de tipo BMP 24 bits– en su interior) y un convertidor de nivel 3,3V<->5V para trabajar sin problemas con placas Arduino. El controlador que lleva incorporado es sin embargo, diferente: se trata del ILI9340 de Ilitek; esto hace que las librerías necesarias para programar este módulo sean, además de la "Adafruit GFX" ya conocida, la "Adafruit ILI9340" (https://github.com/adafruit/Adafruit_ILI9340). Las conexiones son idénticas a las del producto mencionado en el párrafo anterior (de hecho, hasta la nomenclatura de cada pin del módulo es igual).

EL MUNDO GENUINO-ARDUINO

Otro módulo casi idéntico al descrito en el párrafo anterior es el "**ITDB02-2.2S**", en este caso distribuido por IteadStudio (empresa que también ofrece, por si nos interesa algo más pequeño, el módulo "**ITDB02-1.8**", el cual dispone de una pantalla TFT de 128x160 píxeles y 1,8 pulgadas controlada en este caso por el chip ST7735).

Finalmente, destacaremos un módulo muy interesante ofrecido por la empresa 4DSystems, el "**uLCD-144-G2**". Este módulo ofrece, además de una pantalla TFT de 128x128 píxeles y 1,44 pulgadas capaz de mostrar más de 65000 colores, un zócalo micro-SD y, como novedad, un par de pines digitales de entrada/salida ("IO1" y "IO2") que permiten conectarlo directamente a sensores y/o actuadores sin necesidad de utilizar ninguna placa Arduino (eso sí, se tiene que alimentar siempre a tierra y a una fuente de 5V). Este comportamiento autónomo es posible gracias a la presencia en el módulo de un microcontrolador llamado Goldelox2 (fabricado por la misma 4DSystems), el cual, además de encargarse de la entrada/salida del módulo, es el responsable final real de toda la gestión gráfica de la pantalla. El microcontrolador Goldelox2 es programable mediante un lenguaje propio llamado 4DGL; los códigos 4DGL han de ser escritos usando un software llamado "Workshop IDE" (el cual incluye tanto el entorno de desarrollo propiamente dicho como un diseñador gráfico de interfaces) y han de ser cargados desde nuestro computador en el Goldelox2 (una vez ya compilados) mediante un adaptador USB-Serie especial llamado "**uUSB-PA5**". Desgraciadamente, este software es de pago y solo para Windows pero hay una alternativa: si conectamos este módulo (a través de sus pines "TX" y "RX") a una placa Arduino, podemos conseguir que sea esta quien controle el dibujado de las figuras (líneas, rectángulos, círculos, etc.), la visualización de fotografías, la impresión del texto por pantalla, la grabación de datos en la tarjeta microSD o, incluso, la emisión de sonido simplemente ejecutando un sketch que envíe a Goldelox2 (con `Serial.write()`; a través del canal serie establecido) los comandos hexadecimales adecuados (consultables en el documento "Serial Command Set Reference Manual", descargable de su web). Si escribir comandos hexadecimales nos pareciera algo engoroso, podemos usar como alternativa una librería (desarrollada por 4DSystems) llamada "Goldelox Serial Arduino Library" (<https://github.com/4dsystems/Goldelox-Serial-Arduino-Library>) que ofrece múltiples funciones para poder realizar las operaciones gráficas básicas de una forma mucho más intuitiva.

Existe la posibilidad de utilizar el módulo anterior como shield incrustable sobre una placa Arduino si le acoplamos un adaptador llamado "**Arduino Adaptor Shield**" y también es posible adquirir un kit incluyendo el "uLCD-144-G2" y el "Arduino Adaptor Shield" todo en uno: el producto llamado "**uLCD-144-G2-AR**".

Otro módulo TFT interesante distribuido por 4DSystems es el producto "uLCD-220RD" (o "uLCD-220RD-AR" si se adquiere junto con el adaptador "Arduino Adaptor Shield"); tiene la particularidad de ofrecer una pantalla redonda (de 220x200, 1,38" y ~65000 colores) controlada por el chip Diablo16, el cual también puede ser gestionado mediante comandos hexadecimales a través del canal serie. También dispone de zócalo micro-SD y de varios pines digitales de entrada/salida.

Shields y módulos que incorporan pantallas TFT táctiles

Pantallas TFT táctiles resistivas vs. pantallas TFT táctiles capacitivas

En el mercado existen básicamente dos tipos de pantallas TFT táctiles: las basadas en tecnología resistiva y las basadas en tecnología capacitiva. De forma muy esquemática, podemos decir que ambas están formadas por dos capas conductoras separadas entre sí por un material aislante; no obstante, mientras que en el caso de las TFT resistivas su capa inferior (la que se ilumina y muestra las imágenes) está compuesta por una superficie de cristal cubierta de un material conductor transparente y su capa superior (la que recibe la interacción del usuario) está compuesta por una superficie de ese mismo material conductor cubierto por un plástico flexible externo –también transparente–, en el caso de las TFT capacitivas, aunque su capa inferior está igualmente compuesta de una superficie de cristal cubierta de material conductor transparente, su capa superior, en cambio, está compuesta de ese mismo material conductor pero cubierto ahora por cristal (no flexible, obviamente).

El funcionamiento de las TFT resistivas es el siguiente: al ser su capa superior flexible, bajo una presión lo suficientemente elevada ejercida sobre ella por cualquier instrumento (un dedo, un punzón, etc.), esta se deforma lo necesario para entrar en contacto con la capa inferior, cerrándose entonces el circuito eléctrico formado por el material conductor de ambas capas; de esta manera es posible identificar el punto concreto de la pulsación como el punto donde se produce ese cierre. El funcionamiento de las TFT capacitivas, en cambio, se basa en el hecho de que el cuerpo humano es conductor (es decir, permite el paso de electricidad a través de él); esto hace que tan solo con tocar su capa superior sin ejercer ninguna presión adicional (recordemos que esta capa no es flexible), cambie el campo electrostático de la pantalla de tal forma que se pueda localizar el punto de contacto.

La diferencia principal entre ambos tipos de pantallas es que mientras las resistivas necesitan un cierto grado de presión para reconocer la interacción del usuario, en las capacitivas basta un simple contacto. A cambio, las primeras son capaces de reaccionar ante un instrumento puntero hecho de cualquier material

EL MUNDO GENUINO-ARDUINO

(dedos, guantes, un lápiz,...) mientras que las segundas solo pueden reaccionar ante un instrumento conductor (dedos, un punzón especializado, etc.). Otra diferencia importante es que las resistivas suelen ser menos brillantes que las capacitivas (debido al propio diseño y materiales empleados en su fabricación) aunque, por el mismo motivo, también son más baratas. Finalmente, otra diferencia (relevante en determinadas aplicaciones) es que las pantallas resistivas no son capaces de detectar varios puntos de contacto simultáneos (lo que se llama "multitouch"), a diferencia de las pantallas capacitivas, que sí lo son.

De tecnología resistiva

Adafruit distribuye su "2,8" TFT Touch Shield" como producto nº 1651. Este shield incorpora un zócalo para insertar una tarjeta microSD (que ha de estar formateada en FAT16 o FAT32 para poder guardar imágenes –en formato BMP 24 bits, eso sí– en ella) y una pantalla TFT de 240x320 píxeles y 2,8 pulgadas de diagonal que ofrece iluminación de fondo y que permite mostrar 2^{18} colores diferentes. El controlador de la pantalla que lleva incorporado (el ILI9341 de Ilitek) trabaja a 3,3V (con un consumo aproximado de 100mA según el uso) pero el shield incorpora un regulador de voltaje interno que permite no preocuparnos de este tema. Por otro lado, el sensor resistivo adherido a toda la pantalla (que convierte a esta en un dispositivo táctil) está controlado por el chip STMPE610 de ST, también integrado en el shield. Por tanto, para usar este producto tan solo tenemos que encazarlo sobre la placa Arduino y listo.

Tanto la pantalla propiamente dicha (a través del chip ILI9341) como el sensor resistivo (a través del chip STMPE610) y el zócalo microSD se comunican con la placa Arduino mediante el protocolo SPI. Concretamente, aparte de los pines nº 13 (SCLK), nº 12 (MISO) y nº 11 (MOSI), que son compartidos indistintamente por la pantalla, por el sensor y por el zócalo microSD, la pantalla monopoliza el uso del pin digital nº 9 (correspondiente a la línea D/C, encargada de indicar a la pantalla si las señales transmitidas por la línea MOSI son o comandos de control o bien información a mostrar) y del pin nº 10 (correspondiente a su línea CS particular), el sensor monopoliza el uso del pin nº 8 (correspondiente a su línea CS particular) y el zócalo microSD monopoliza el uso del pin nº 4 (correspondiente a su línea CS particular). Esto quiere decir que aún quedan disponibles los pines digitales nº 2, 3, 5, 6 y 7 (el nº 4 estaría libre también si no se usara tarjeta microSD), y todos los pines analógicos.

Para poder trabajar con este shield, necesitaremos instalar tres librerías: la propia del chip controlador de la pantalla, llamada "Adafruit ILI9341" (https://github.com/adafruit/Adafruit_ILI9341), necesaria para poder reconocer y manejar la pantalla a nivel interno pero raramente utilizada directamente por

nosotros, la –ya conocida– librería "Adafruit GFX", verdadera responsable del dibujado de los píxeles, líneas, rectángulos, triángulos, círculos, imágenes, texto..., y la librería "Adafruit STMPE610" (https://github.com/adafruit/Adafruit_STMPE610), necesaria para aprovechar la capacidad táctil de la pantalla al ofrecer de forma muy sencilla la posibilidad de detectar la "x", la "y" y la "z" –presión– del punto de contacto. Todas estas librerías vienen con ejemplos de código muy bien documentados, tanto para mostrar las capacidades gráficas de la pantalla como las de respuesta táctil.

No obstante, si lo que deseamos es disponer de una pantalla TFT táctil resistiva montada en forma de módulo independiente en vez de como shield (para así poderla conectar a la placa Arduino mediante cables de tal forma que los pinos-hembra no usados de esta puedan estar disponibles sin tener nada por encima que los tape) el producto nº 1770 de Adafruit es la opción más parecida al shield anterior, ya que ofrece la misma pantalla (con el mismo chip controlador ILI9341 y su correspondiente regulador 3,3V<->5V incorporado) y también un zócalo microSD. La mayor diferencia entre ambos productos es que este módulo, a diferencia del shield anterior, no incorpora ningún chip específico para controlar el sensor táctil (chip que, por otro lado, se podría adquirir de forma independiente como producto nº 1571); esto hace que para detectar los puntos de contacto tengamos que utilizar obligatoriamente cuatro conexiones de nuestra placa Arduino en exclusiva (dos digitales y dos analógicas, tal como veremos enseguida).

Existen dos formas de conectar la pantalla TFT del módulo descrito en el párrafo anterior a una placa Arduino: en modo "8-bit" o bien mediante SPI; aquí solo detallaremos esta última al requerir menos cables y ser más cómoda de implementar. Cada una de estas dos formas requiere emplear una ristra de conectores pinos-macho diferente (a soldar, ya que no vienen soldados de fábrica), ubicadas respectivamente en sendos laterales del módulo (y bien seriografiadas para evitar cualquier confusión). Concretamente, si usamos el protocolo SPI, las conexiones a realizar son las siguientes:

Módulo TFT	Placa Arduino UNO
GND	GND
3-5V	5V
CLK	Pin digital nº 13
MISO	Pin digital nº 12 (si se usa tarjeta SD)
MOSI	Pin digital nº 11
CS	Pin digital nº 10 (para la pantalla)
D/C	Pin digital nº 9

EL MUNDO GENUINO-ARDUINO

Módulo TFT	Placa Arduino UNO
Lite	Un pin PWM o 5V (para luz de fondo)
Y+	Un pin analógico (a definir en el código)
X+	Un pin digital (a definir en el código)
Y-	Un pin digital (a definir en el código)
X-	Un pin analógico (a definir en el código)
IM3	3,3V
IM2	3,3V
IM1	3,3V
Card CS	Pin digital nº 4 (si se usa tarjeta SD)

Tal como se puede ver en la tabla anterior, además de las conexiones SPI de la placa Arduino UNO a la pantalla y al zócalo microSD (y la conexión de alimentación y tierra común, por supuesto), también aparece la conexión a la luz de fondo ("lite") y, sobre todo, la conexión al sensor táctil, la cual, al no haber ahora chip controlador, monopoliza –tal como hemos comentado en un párrafo anterior– cuatro pines más (Y+, X+, Y- y X-): dos de tipo digital (entrada) y dos de tipo analógico, pines cuyo número deberá ser indicado convenientemente en nuestro sketch. Notar además que aún necesitamos conectar tres líneas más (concretamente las marcadas como "IM1", "IM2" y "IM3") a la señal de 3,3V; esto es para activar en el módulo la capacidad de utilizar el protocolo SPI, ya que sin estas tres líneas a 3,3V, el módulo solamente podrá comunicarse en modo "8-bit".

Para poder trabajar con este módulo, necesitaremos instalar tres librerías: además de la omnipresente librería "Adafruit GFX" y de la librería "Adafruit Touch Screen" (<https://github.com/adafruit/Touch-Screen-Library>), necesaria para aprovechar la capacidad táctil de la pantalla sin usar ningún chip controlador), si trabajamos en modo "8-bits", necesitaremos la librería "Adafruit TFTLCD" (<https://github.com/adafruit/TFTLCD-Library>) y si trabajamos usando SPI necesitaremos la librería propia del chip controlador de la pantalla, la "Adafruit ILI9341" (ya mencionada en párrafos anteriores). Todas estas librerías vienen con ejemplos de código muy bien documentados, tanto para mostrar las capacidades gráficas de la pantalla como sus capacidades de respuesta táctil.

Otros módulos muy similares al recién descrito (y también distribuidos por Adafruit) son su producto **nº 2478** y su producto **nº 2050**. La diferencia más importante entre ellos es el tamaño de la pantalla TFT táctil resistiva que incorporan: en el caso del producto nº 2478 se trata de una pantalla de 2,4 pulgadas y 320x240 píxeles de resolución (controlada por el mismo chip ILI9341), mientras que en el caso del producto nº 2050 se trata de una pantalla de 3,5 pulgadas y 320x480 píxeles de

resolución (controlada, eso sí, por el chip HXD8357D, lo que conlleva que, en vez de la "Adafruit ILI9341", se tenga que utilizar la librería "Adafruit HXD8357" https://github.com/adafruit/Adafruit_HX8357_Library).

Otro módulo (que no shield) con pantalla táctil resistiva muy interesante es el "**Smart GPU 2**" de Vizictechnologies, disponible en varios tamaños de pantalla (320x240 y 2,4", 480x320 y 3,5", 480x272 y 4,2") pero todas con capacidad para mostrar más de 250000 colores. Este módulo incorpora además un zócalo microSD donde la tarjeta a insertar –que ha de estar formateada en FAT16 o FAT32– podrá almacenar no solo fotografías de tipo BMP sino también de tipo JPG (gracias al controlador gráfico presente en dicho módulo); este módulo también es capaz de mostrar vídeos (siempre y cuando estos estén en un formato especial llamado "VID"; se puede transformar cualquier vídeo a este formato mediante un programa de computador llamado "Video Converter", descargable de la web del producto) y de reproducir audio estéreo (a través de los pines "AL" –salida canal izquierdo– y "AR" –salida canal derecho–). Tanto la pantalla como el sensor táctil como la tarjeta microSD se gestionan mediante una librería propia, descargable de su web, junto con varios ejemplos de código. Internamente se comunica vía serie con el receptor/transmisor TTL-UART del ATmega328P, por lo que, en realidad, todas las instrucciones de esa librería (de tipo *milcd.drawLine()*, *milcd.drawCircle()*, *milcd.imageSD()*, *milcd.string()*, etc.) internamente no son más que simples comandos hexadecimales enviados vía serie. Los únicos pines imprescindibles que se han de utilizar para conectar este módulo con una placa Arduino son el de alimentación (¡ha de ser de 3,3V!), tierra, RESET, TX (al RX del Arduino) y RX (al TX del Arduino). Finalmente, también es interesante considerar el producto complementario "**SmartSHIELD**", un shield que proporciona una manera muy sencilla de conectar el "Smart GPU 2" a una placa Arduino.

IteadStudio, por su parte, también distribuye varios módulos TFT táctiles resistivos (todos con zócalo SD incorporado), entre los cuales podemos encontrar el módulo "**ITDB02-2.4E**" (con un sensor táctil basado en el chip TSC2046 y con una pantalla de 2,4 pulgadas, 320x240 píxeles de resolución, capaz de mostrar 65000, controlada por el chip S6D1121 y funcionando en modo "8-bits"... ojo porque es diferente del "ITDB02-2.4S"), el "**ITDB02-2.8**" (similar al anterior pero con una pantalla de 2,8 pulgadas controlada por el chip ILI9325DS), el "**ITDB02-3.2S**" (similar al anterior pero con una pantalla de 3,2 pulgadas controlada por el chip SSD1289), el "**ITDB02-4.3**" (similar al anterior pero con una pantalla de 4,3 pulgadas y 272x480 píxeles controlada por el chip SSD1963) o el "**ITDB02-5.0**" (similar al anterior pero con una pantalla de 5 pulgadas y 800x480 píxeles de resolución), entre otros. IteadStudio también distribuye un shield llamado "**ITDB02 Arduino shield**" que

EL MUNDO GENUINO-ARDUINO

permite acoplar sin problemas cualquiera de los módulos anteriores sobre una placa Arduino UNO. Todos ellos pueden ser programados mediante la librería "UTFT", descargable desde <http://www.rinkydinkelectronics.com/library.php>. La librería "UTFT", de hecho, puede ser utilizada para trabajar con módulos TFT de muchos otros fabricantes (como por ejemplo los módulos de la serie "TFT01" de Elecfreaks, por poner un ejemplo), así que es probable que la volvamos a ver en otras ocasiones. No obstante, esta librería tan solo maneja el aspecto gráfico de las pantallas: para utilizar su capacidad táctil, necesitamos descargar e instalar además la librería "UTouch", también disponible en la web de RinkydinkElectronics. Asimismo, para cargar las imágenes ubicadas dentro de la tarjeta microSD –que en este caso deberá estar formateada obligatoriamente en FAT16–, será necesario usar dos librerías extra más: la "UTFT_tinyFAT" (la utilizada directamente en nuestros sketches) y la "tinyFAT" (base a nivel interno de la anterior), ambas disponibles también en la web de RinkydinkElectronics. Finalmente, cabe destacar la existencia de las librerías "UTFT.Buttons" y "UTFT_Geometry": la primera añade la capacidad de mostrar botones en pantalla para interactuar con el usuario de forma sencilla y la segunda incluye varias funciones de pintado de figuras geométricas.

IteadStudio distribuye además varios shields TFT táctiles completos (con pantalla y zócalo SD incorporados), como el "ITEAD 2.4 TFT LCD Touch shield", el "ITEAD 2.8 TFT LCD Touch shield" o el "ITEAD 3.2 TFT LCD Touch shield". Desgraciadamente, el último solo es compatible con Arduino MEGA y los dos primeros no son compatibles con la placa Arduino UNO (debido a que no es posible el encaje físico al ser el zócalo USB de la placa –de tipo B, recordemos– demasiado grande) sino con una placa clónica propia de IteadStudio llamada "**Iteaduino**" (donde se ha sustituido dicho zócalo por un conector hembra mini-USB, mucho más pequeño). Estos shields no ofrecen una librería específica para Arduino: tan solo códigos de ejemplo (con funciones reutilizables, eso sí).

Otro shield que incluye una pantalla TFT táctil resistiva es el distribuido por SeeedStudio con código de producto **SLD10261P**; al igual que los anteriores, consta de un zócalo microSD, de un sensor táctil resistivo y de una pantalla (en este caso, de 2,8 pulgadas y 320x240 píxeles, con capacidad para mostrar 65000 colores, controlada por el chip ILI9341 y funcionando vía SPI). Para pintar dibujos, textos, figuras, etc., y también para mostrar imágenes guardadas en la tarjeta microSD se ha de utilizar la librería gráfica descargable de https://github.com/Seeed-Studio/TFT_Touch_Shield_V2; si se quiere hacer uso, además, de las capacidades táctiles de la pantalla, se ha de emplear la librería suplementaria específica https://github.com/Seeed-Studio/Touch_Screen_Driver.

De tecnología capacitiva

Adafruit distribuye su "2,8" TFT Touch Shield" como producto nº 1947. Este shield incorpora un zócalo para insertar una tarjeta microSD (que ha de estar formateada en FAT16 o FAT32 para poder guardar imágenes –en formato BMP 24 bits, eso sí– en ella) y una pantalla TFT de 240x320 píxeles y 2,8 pulgadas de diagonal que ofrece iluminación de fondo y que permite mostrar 2^{18} colores diferentes. El controlador de la pantalla que lleva incorporado (el ILI9341 de Ilitek) trabaja a 3,3V (con un consumo aproximado de 100mA según el uso) pero el shield incorpora un regulador de voltaje interno que permite no preocuparnos de este tema. Por otro lado, el sensor capacitivo adherido a toda la pantalla (que convierte a esta en un dispositivo táctil) está controlado por el chip FT2606 de Focaltech, también integrado en el shield. Por tanto, para usar este producto tan solo tenemos que encazarlo sobre la placa Arduino y listo.

Tanto la pantalla propiamente dicha (a través del chip ILI9341) como el zócalo microSD se comunican con la placa Arduino mediante el protocolo SPI; en cambio, el sensor capacitivo (a través del chip FT2606), se comunica vía I²C. Más en concreto, los pines digitales nº 13 (SCLK), nº 12 (MISO) y nº 11 (MOSI) son compartidos indistintamente por la pantalla y por el zócalo microSD, mientras que la pantalla monopoliza el uso del pin digital nº 9 (correspondiente a la línea D/C) y del pin nº 10 (correspondiente a su línea CS particular) y el zócalo microSD monopoliza el uso del pin digital nº 4 (correspondiente a su línea CS particular). El sensor capacitivo, por su parte, utiliza –sin monopolizar– los pines analógicos nº 4 (SDA) y nº 5 (SCL). Esto quiere decir que aún quedan disponibles los pines digitales nº 2, 3, 5, 6, 7 y 8 (el nº 4 estaría libre también si no se usara tarjeta microSD), además de todos los pines analógicos.

Para poder trabajar con este shield, necesitaremos instalar tres librerías: la propia del chip controlador de la pantalla, llamada "Adafruit ILI9341" (https://github.com/adafruit/Adafruit_ILI9341, necesaria para poder reconocer y manejar la pantalla a nivel interno pero raramente utilizada directamente por nosotros), la –ya conocida– librería "Adafruit GFX" (verdadera responsable del dibujado de los píxeles, líneas, rectángulos, triángulos, círculos, imágenes, texto...) y la librería "Adafruit FT6206" (https://github.com/adafruit/Adafruit_FT6206_Library, necesaria para aprovechar la capacidad táctil de la pantalla al ofrecer de forma muy sencilla la posibilidad de detectar la "x" y la "y" del punto de contacto). Todas estas librerías vienen con ejemplos de código muy bien documentados, tanto para mostrar las capacidades gráficas de la pantalla como sus capacidades de respuesta táctil.

EL MUNDO GENUINO-ARDUINO

No obstante, si lo que deseamos es disponer de una pantalla TFT táctil capacitiva montada en forma de módulo independiente en vez de como shield (para así poderla conectar a la placa Arduino mediante cables de tal forma que los pines-hembra no usados de esta puedan estar disponibles sin tener nada por encima que los tape) el producto nº 2090 de Adafruit es la opción más parecida al shield anterior, ya que ofrece la misma pantalla (con el mismo chip controlador ILI9341 y su correspondiente regulador 3,3V<->5V incorporado), el mismo zócalo microSD y el mismo sensor capacitivo (controlado asimismo por el chip FT2606). Independientemente de este último, que siempre se comunica vía I²C, existen dos formas de conectar la pantalla TFT (propriamente dicha): en modo "8-bit" o bien mediante SPI; aquí solo detallaremos esta última al requerir menos cables y ser más cómoda de implementar. Cada una de estas dos formas requiere emplear una ristra de conectores pines-macho diferente (a soldar, ya que no vienen soldados de fábrica), ubicadas respectivamente en sendos laterales del módulo (y bien seriagrfiadas para evitar cualquier confusión). Concretamente, si usamos el protocolo SPI, las conexiones a realizar son las siguientes:

Módulo TFT	Placa Arduino UNO
GND	GND
Vin	5V
CLK	Pin digital nº 13
MISO	Pin digital nº 12 (si se usa tarjeta SD)
MOSI	Pin digital nº 11
CS	Pin digital nº 10 (para la pantalla)
DC	Pin digital nº 9
Lite	Un pin PWM o 5V (para luz de fondo)
SDA	Pin analógico nº 4
SCL	Pin analógico nº 5
IM3	3,3V
IM2	3,3V
IM1	3,3V
Card CS	Pin digital nº 4 (si se usa tarjeta SD)

Tal como se puede ver en la tabla anterior, además de las conexiones SPI de la placa Arduino UNO a la pantalla y al zócalo microSD (y la conexión de alimentación y tierra común, por supuesto), también aparece la conexión a la luz de fondo ("lite") y la conexión I²C al sensor táctil. Notar además que aún necesitamos conectar tres líneas más (concretamente las marcadas como "IM1", "IM2" y "IM3") a la señal de 3,3V; esto es para activar en el módulo la capacidad de utilizar el protocolo SPI, ya que sin estas tres líneas a 3,3V, el módulo solamente podrá comunicarse en modo "8-bit".

Para poder trabajar con este módulo, necesitaremos instalar tres librerías: además de la omnipresente librería "Adafruit GFX" y de la ya mencionada librería "Adafruit FT6206", si trabajamos en modo "8-bits", necesitaremos la librería "Adafruit TFTLCD" mientras que si trabajamos usando SPI necesitaremos la librería propia del chip controlador de la pantalla, la "Adafruit ILI9341" (ambas ya mencionadas también en párrafos anteriores).

USO DE PANTALLAS OLED

Las pantallas OLED

Aparte de las pantallas de cristal líquido (LCDs "clásicas" o TFT), existen otros tipos de pantalla construidas con diferentes tecnologías y materiales. Uno de esos "otros" tipos son las pantallas OLED. Al igual que las TFT, las pantallas OLED permiten dibujar gráficos a todo color: desde simples píxeles, rectángulos o círculos hasta fotografías e incluso vídeo (de hecho, la mayoría de módulos OLED también disponen de un zócalo para tarjetas de memoria micro-SD para almacenar tanto datos como las imágenes, animaciones o vídeos a reproducir). No obstante, el hecho de estar compuestas por un tipo especial de LEDs (los llamados "orgánicos", de ahí su nombre) hace que las pantallas OLED sean más delgadas, ligeras y flexibles, ofrezcan más contraste y brillo (pueden utilizarse incluso a plena luz del sol), realicen un menor consumo (por, entre otras cosas, no necesitar luz de fondo) y tengan mayor ángulo de visión que las pantallas TFT. Por contra, la relativamente rápida degradación de los materiales con los que se fabrican ha limitado de momento su uso (aunque se está investigando para dar solución a estos problemas).

Arduino no ofrece ninguna pantalla OLED como producto oficial y, por tanto, tampoco mantiene ninguna librería relacionada con este tipo de pantallas. No obstante, se ha considerado útil incluir en este libro el siguiente apartado para que el lector tenga una perspectiva más amplia de las posibilidades que puede tener al emplear este tipo de pantallas en sus circuitos.

Módulos OLED de 4DSystems

El fabricante 4DSystems ofrece un conjunto de módulos con pantalla OLED integrada (concretamente, pantallas OLED de tipo "Matriz Pasiva", o "PMOLED") que básicamente se diferencian en la resolución y tamaño de dicha pantalla. En concreto, podemos listar el "**uOLED-96-G2**" (con pantalla de 96x64 píxeles y 0,96 pulgadas de diagonal), el "**uOLED-128-G2**" (con pantalla de 128x128 píxeles y 1,5 pulgadas) y el "**uOLED-160-G2**" (con pantalla de 160x128 píxeles y 1,7 pulgadas); todas estas pantallas pueden mostrar la misma cantidad de colores (más de 65000). Todos estos

EL MUNDO GENUINO-ARDUINO

módulos ofrecen además un par de pines digitales de entrada/salida ("IO1" y "IO2") que permiten conectarlos directamente a sensores y/o actuadores sin necesidad de utilizar ninguna placa Arduino (eso sí, se tienen que alimentar siempre a tierra y a una fuente de 5V).

Este comportamiento autónomo es posible gracias a la presencia en el módulo de un microcontrolador llamado Goldelox2 (fabricado por la misma 4DSystems), el cual, además de encargarse de la entrada/salida del módulo, es el responsable final real de toda la gestión gráfica de la pantalla. El microcontrolador Goldelox2 es programable mediante un lenguaje propio llamado 4DGL; los códigos 4DGL han de ser escritos usando un software llamado "Workshop IDE" (el cual incluye tanto el entorno de desarrollo propiamente dicho como un diseñador gráfico de interfaces) y han de ser cargados desde nuestro computador en el Goldelox2 (una vez ya compilados) mediante un adaptador USB-Serie especial llamado "**uUSB-PAS**".

Desgraciadamente, este software es de pago y solo para Windows pero hay una alternativa: si conectamos este módulo (a través de sus pines "TX" y "RX") a una placa Arduino, podemos conseguir que sea esta quien controle el dibujado de las figuras (líneas, rectángulos, círculos, etc.), la visualización de fotografías, la impresión del texto por pantalla, la grabación de datos en la tarjeta micro-SD o, incluso, la emisión de sonido simplemente ejecutando un sketch que envíe a Goldelox2 (con `Serial.write()`; a través del canal serie establecido) los comandos hexadecimales adecuados, consultables en el documento "Serial Command Set Reference Manual", descargable de su web. De todas formas, si escribir comandos hexadecimales nos pareciera algo engorroso, podemos utilizar como alternativa una librería (desarrollada por 4DSystems) llamada "Goldelox Serial Arduino Library" (<https://github.com/4dsystems/Goldelox-Serial-Arduino-Library>) que ofrece múltiples funciones para poder realizar las operaciones gráficas básicas de una forma mucho más intuitiva.

Por otro lado, también existe la posibilidad de utilizar cada uno de los módulos anteriores como shields incrustables sobre una placa Arduino si les acoplamos un adaptador llamado "**Arduino Adaptor Shield**"; también es posible adquirir un kit incluyendo el módulo OLED deseado y el "Arduino Adaptor Shield" todo en uno: son los productos "**uOLED-96-G2-AR**", "**uOLED-128-G2-AR**" y "**uOLED-160-G2-AR**", respectivamente.

Módulos OLED de Adafruit

Adafruit distribuye cuatro módulos OLED monocromos: el producto **nº 661** (con pantalla de 128x32 píxeles y 0,91 pulgadas de diagonal), el producto **nº 931** (con

pantalla de también 128x32 píxeles y 0,91 pulgadas de diagonal), el producto nº 938 (con pantalla de 128x64 píxeles y 1,3 pulgadas) y el producto nº 326 (con pantalla de 128x64 píxeles y 0,96 pulgadas). Todos incluyen el mismo chip (el SSD1306 de Solomon Systech), el cual puede comunicarse con la placa Arduino de dos maneras: vía SPI (más rápido) o vía I²C (requiere menos cables). No obstante, debido a sus diseños particulares, el producto nº 661 solamente permite emplear SPI mientras que el producto nº 931 solamente permite emplear I²C. En cambio, tanto el producto nº 938 como el nº 326 sí permiten seleccionar el protocolo a usar: más en concreto, de entrada ambos módulos vienen preparados para comunicarse por defecto mediante el protocolo SPI (así que no habría que hacer nada en particular si es este el método elegido) pero si se desea utilizar el protocolo I²C, simplemente bastará con deshacer (soldando) el corte existente en dos de las zonas de metal ubicadas en el dorso del módulo en cuestión: concretamente los etiquetados como "SJ1" y "SJ2".

Los cuatro módulos anteriores incorporan internamente un elevador de tensión para convertir los 5V de los pines de Arduino en hasta 12V y poder alimentar así los LEDs de la pantalla (con un consumo total de aproximadamente 40mA), pero, no obstante, su circuitería electrónica solo puede ser alimentada con 3,3V para funcionar correctamente. Afortunadamente, todos incorporan internamente un regulador de tensión general que rebaja los 5V a 3,3V, por lo que podremos conectarlos a nuestra placa Arduino sin problemas.

En el caso del producto nº 661 y de los productos nº 938 y nº 326 funcionando vía SPI, el cableado necesario es el siguiente: el pin "GND" ha de ir a tierra, el pin "Vin" ha de ir al pin de 5V de la placa Arduino, "DATA" debe ir al pin digital nº 9, "CLK" al pin digital nº 10, "D/C" al pin digital nº 11, "CS" al pin digital nº 12 y "RST" al pin digital nº 13. En el caso del producto nº 931 y de los productos nº 938 y nº 326 funcionando vía I²C, el pin "GND" ha de ir a tierra, el "Vin" al pin de 5V, el pin "SDA" al pin de entrada analógica nº 4 (en el modelo UNO), el pin "SCL" al pin de entrada analógica nº 5 y el pin "RST" al pin digital nº 4.

Una vez hechas las conexiones, para poder utilizar cualquiera de los módulos anteriores deberemos descargar e instalar, además de la ya conocida "Adafruit GFX", la librería "Adafruit SSD1306" (https://github.com/adafruit/Adafruit_SSD1306). Esta librería incluye códigos de ejemplo muy interesantes para estudiar (accesibles dentro del menú "File->Sketchbook->Libraries->Adafruit_SSD1306" del IDE Arduino); estos ejemplos muestran cómo mostrar diversos textos, imágenes, rectángulos, círculos y líneas por la pantalla en diferentes posiciones y tamaños.

Un problema que tienen todos los módulos monocromos anteriores es que carecen de zócalo micro-SD o de cualquier otro tipo de sistema de almacenamiento para alojar ningún fichero de imagen. Esto obliga a tener que usar el "truco" descrito

EL MUNDO GENUINO-ARDUINO

anteriormente en el cuadro titulado "Breve nota sobre la visualización de imágenes al vuelo" para poder visualizar fotografías en ellos.

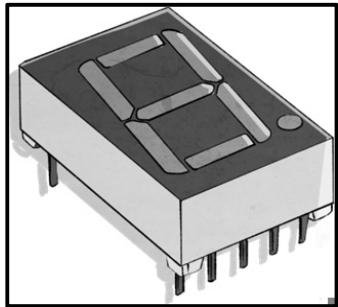
Por otro lado, Adafruit también ofrece tres módulos OLED con pantalla RGB capaces de mostrar fotografías de más de 65000 colores (en formato BMP 24-bits) y con un zócalo microSD integrado para poder almacenar esas imágenes. Estos tres módulos son el producto **nº 684** (con pantalla de 96x64 píxeles y 0,96 pulgadas de diagonal), el producto **nº 1673** (con pantalla de 128x96 píxeles y 1,27 pulgadas de diagonal) y el producto **nº 1431** (con pantalla de 128x128 píxeles y 1,5 pulgadas de diagonal). Aunque su circuitería interna funciona a 3,3V, todos estos módulos pueden ser conectados directamente a nuestra placa Arduino sin problemas porque incorporan internamente un regulador de tensión general (así como también un elevador de tensión para poder alimentar los LEDs de la pantalla). Se comunican vía SPI, por lo que, al igual que los módulos monocromo SPI anteriores, tan solo requieren cinco cables ("clock", "data", "chip select", "data/command" y "reset") además de alimentación y tierra. Todos estos módulos están controlados por el chip SSD1331, por lo que para ser gestionados desde un sketch Arduino es necesario utilizar la librería "Adafruit SSD1331" (<https://github.com/adafruit/Adafruit-SSD1351-library>), además de la ya conocida "Adafruit GFX".

Finalmente, destacaremos la existencia del producto **nº 823** de Adafruit, que consiste en una pantalla OLED "de caracteres" (esto es, similar –en utilidad, tamaño y aspecto– a una pantalla LCD de caracteres 16x2 "clásica"). Las ventajas de la tecnología OLED ya se han comentado: mayor contraste, mayor ángulo de legibilidad y menor consumo, por lo que con este producto conseguimos aunar estas ventajas con la facilidad y universalidad de uso de las pantallas LCD convencionales; por ejemplo, a nivel de conexiones –ya sean en modo "4-bit" o "8-bit"–, son idénticas (exceptuando la inexistencia de la conexión de la luz de fondo y la obligatoriedad de la conexión "r/w"). Eso sí, al estar controlada esta pantalla por el chip WS0010 (en vez del HD44780 típico de las LCD), debemos emplear una librería diferente de la "LiquidCrystal" oficial (aunque muy parecida) llamada "Adafruit CharacterOLED" (https://github.com/ladyada/Adafruit_CharacterOLED). Un producto similar (de hecho, se programa con la misma librería) es el **nº 11987** de Sparkfun.

USO DE OTRAS PANTALLAS

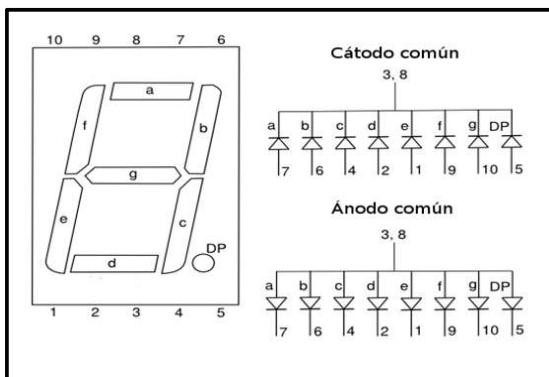
Arduino no mantiene oficialmente ninguna librería relacionada con los tipos de pantallas mencionados a continuación. No obstante, se ha considerado útil incluirlas en este libro para que el lector tenga una perspectiva más amplia sobre las posibilidades que ofrecen este tipo de pantallas en sus circuitos.

7-segmentos



Se llama "display 7-segmentos" a un encapsulado transparente compacto que contiene 7 (u ocho) LEDs en su interior conectados entre sí de tal manera que encendiendo algunos de ellos y apagando otros podemos mostrar la forma de diferentes cifras. Tal vez su uso pueda parecer redundante al lado de una LCD, pero si en nuestro proyecto solamente deseamos ciertos símbolos y/o se necesita una visibilidad realmente grande, los displays 7-segmentos son la mejor opción. Además, consumen mucha menos corriente que un LCD con luz de fondo.

Un display 7-segmentos típico suele tener diez pines ubicados en dos filas de cinco (e identificados en el datasheet mediante un número, del 1 al 10): siete de estos pines (para saber cuáles son en concreto hay que consultar el datasheet) están unidos internamente a sendos LEDs (identificados en el datasheet como "a", "b", "c", "d", "e", "f" y "g") y otro pin lo está a un octavo LED (identificado en el datasheet como "DP") que marca el punto decimal. Para poder iluminar cada uno de estos ocho LEDs, el pin correspondiente ha de conectarse a una salida digital (o PWM) diferente de una placa Arduino (eso sí, siempre a través de una resistencia que adapte adecuadamente, dependiendo del color del LED en cuestión, la tensión ofrecida). Los otros dos pines restantes del display (que generalmente están situados en el centro de cada fila y en el datasheet están numerados con las posiciones 3 y 8) o bien han de conectarse ambos a tierra (si el display es de tipo "cátodo común", algo que el fabricante nos debería indicar), o bien ambos a 5V (si el 7-segmentos es de tipo "ánodo común"). El nombre de "cátodo común" o "ánodo común" proviene de la manera en que internamente están conectados los LEDs del display a sus respectivos pines, tal como muestra la siguiente figura, correspondiente a un display típico:



EL MUNDO GENUINO-ARDUINO

Es muy importante tener en cuenta que en el caso de los displays de tipo "cátodo común", para que un LED concreto se ilumine, la salida de la placa Arduino correspondiente ha de estar en valor HIGH pero en el caso de los displays de tipo "ánodo común", en cambio, ocurre al revés: para que un LED se ilumine, la salida correspondiente ha de estar en valor LOW.

Sparkfun distribuye dos modelos de displays 7-segmentos (los dos de tipo ánodo común): el producto **nº 8546** –de color rojo– y el **nº 9191** –de color azul–. Ambos pueden ser controlados perfectamente sin necesidad de ninguna librería externa (ya que para ello tan solo es imprescindible la función *digitalWrite()* propia del lenguaje Arduino, explicada en el siguiente capítulo) pero por comodidad nos puede interesar emplear una librería especializada en este menester llamada "Soda" (<https://github.com/Qtechknow/Arduino-Libraries/tree/master/Soda>).

Sparkfun también distribuye los productos **nº 9480, nº 9481, nº 9482 y nº 9483**; que no son más que conjuntos de cuatro displays 7-segmentos de tipo "ánodo común" funcionando acoplados como una sola unidad (permitiendo así mostrar, por ejemplo, números de hasta cuatro cifras); la única diferencia entre ellos es el color de sus respectivos LEDs. Estos productos requieren 7 pines para iluminar los 7 segmentos que forman una determinada cifra más 4 pines extra para comunicarse con cada una de estas cifras más los pines extra necesarios para iluminar los puntos decimales. Por otro lado, Adafruit también distribuye conjuntos de 4 displays 7-segmentos (aunque no son idénticos –por ejemplo, estos últimos son de tipo "cátodo común"–; para conocer las conexiones exactas recomiendo consultar sus respectivos datasheets); en concreto estamos hablando de los **nº 811, nº 812, nº 813, nº 865 y nº 1001** (cada uno de un color, pero todos de 0,56 pulgadas de altura) y los **nº 1264, nº 1265 y nº 1266** (cada uno de un color, pero todos de 1,2 pulgadas). Destacaremos también los productos **nº 1907, nº 1908, nº 2153, nº 2154, nº 2155 y nº 2156**, todos consistentes en un conjunto de 2 displays de 0,56 pulgadas de tipo "cátodo común" que, en vez de disponer de 7 segmentos, disponen de 14, permitiendo mostrar no solamente cifras numéricas sino cualquier tipo de carácter alfabético, tanto minúsculas como mayúsculas.

Necesidad de aumentar el número de pines de salida

Desgraciadamente, tras lo explicado en los párrafos anteriores, es fácil ver que para manejar un solo display 7-segmentos se necesitan como mínimo 7 salidas digitales de una placa Arduino y para manejar uno de los conjuntos de 4 displays recién descritos no menos de 12. Esto es un gran problema porque, además de propiciar cableados engorrosos, deja a nuestra placa Arduino UNO prácticamente sin pines-hembra disponibles. Para poder solucionar este inconveniente hay varias

opciones: podemos, por ejemplo, aumentar el número de entradas y salidas de una placa Arduino UNO gracias al uso de shields diseñados específicamente para ello (como por ejemplo el "**EZ-Expander shield**" de Nootropic Design –que aporta 16 GPIOs digitales extra, gestionables mediante una librería específica descargable de su web–, el "**I/O Expander Shield**" de LinkSprite –que también aporta 16 GPIOs digitales extra gestionables en este caso directamente a través de la librería "Wire" oficial de Arduino–, el "**IO Expander Shield**" de Numato –que aporta 44 GPIOs digitales y 28 GPIOs analógicos, gestionables también a través de la librería "Wire" oficial de Arduino–, entre otros) o el producto nº **11723** de Sparkfun –que aporta hasta 48 GPIOs que pueden actuar tanto entradas analógicas, entradas digitales o salidas digitales, gestionables mediante una librería específica descargable de su web–. Otro shield interesante es el "**I/O Expansion Shield**" de DFRobot, el cual, aunque no ofrece más pines extra, sí ofrece una manera más conveniente de conectarlos, ya que acompaña cada pin GPIO con su respectivos pines GND y VCC (además de ofrecer otros métodos de comunicación todo en el mismo shield, como un conector I²C, otro SPI, otro para Bluetooth, otro para tarjetas SD, etc.).

Otra opción para disponer de más pines de entrada/salida es utilizar como "intermediario" entre la placa Arduino y el display 7-segmentos un componente electrónico (conectable directamente a una breadboard) cuya función sea actuar precisamente como "aumentador" de entradas y salidas (de hecho, las shields anteriores están basadas en ellos). Esta solución no ayuda a que el cableado sea menos engorroso, pero al menos resuelve el problema de la falta de GPIOs a un coste muy reducido. Existen diferentes tipos de componentes electrónicos que pueden actuar como "aumentadores de pines": los más habituales son los llamados "registros de desplazamiento" (en inglés "shift registers"), los "multiplexadores" y los "expansores E/S I²C o SPI" (un ejemplo de estos últimos son precisamente los "backpacks" para LCDs descritos en un apartado anterior); todos ellos suelen comercializarse en forma de plaquita breakout (o también en formato DIP) pero internamente funcionan de forma muy distinta y por tanto, ofrecen determinadas ventajas o inconvenientes según sus características.

De todas formas, no profundizaremos en ello porque lo más cómodo para nosotros será utilizar directamente shields (o módulos) que ya tengan incorporado de fábrica el "aumentador" y el conjunto de 4 displays 7-segmentos todo integrado, y que ofrezcan una manera sencilla de programarlo. De esto trata el siguiente apartado.

Shields y módulos que incorporan displays 7-segmentos

Nootropic Design ofrece su "**Digit Shield**" listo para acoplar a nuestra placa Arduino, la cual solo monopoliza cuatro pines de entrada/salida digital (los nº 2, 3,

EL MUNDO GENUINO-ARDUINO

4 y 5). Este shield puede ser programado por una librería descargable de su propia web muy sencilla y flexible, la cual contiene diferentes funciones (del tipo *mipantalla.setValue(número)* o *mipantalla.setDigit(posición, cifra)*, entre otras) para mostrarlos valores numéricos de varias maneras.

Otro shield similar es el "**7-segment shield**" de Gravitech, que además de 4 dígitos de 7-segmentos también incorpora "de regalo" un sensor de temperatura, un LED multicolor y una memoria EEPROM extra de 128KBytes. Todo ello monopoliza 4 pines de la placa Arduino. Desgraciadamente, para programarla en nuestro sketch Arduino, debemos utilizar directamente la librería "Wire" oficial de Arduino, ya que no hay ninguna librería que nos oculte la complejidad de la comunicación I²C requerida. De todas formas, nos ofrecen un código de ejemplo para que veamos cómo se ha de hacer.

Si no queremos usar ninguno de los shields anteriores (o similares), también podemos usar módulos con 4 dígitos de 7-segmentos integrados que sean capaces de comunicarse con una placa Arduino de una forma relativamente cómoda (esto es, mediante protocolos como I²C, SPI o TTL-Serie, gracias a llevar incorporado el backpack "conversor a señales 7-segmentos" pertinente. De entre los primeros, podemos destacar el "**I²C 4 digit 7-segment display**" de Gravitech, el cual permite acoplar directamente sus cuatro pines (alimentación 5V, tierra, línea SDA y línea SCL) a una breadboard y es programable usando la librería oficial "Wire" (en su web se proporciona código de ejemplo).

Otro módulo con backpack I²C y con 4 displays 7-segmentos es el "**TWI 7-seg Display**" de Akafugu. Igualmente, solo necesita cuatro cables para ser conectado a una placa Arduino: alimentación (puede ser tanto 3,3V como 5V), tierra, línea SDA y línea SCL. Para trabajar con él se ha de utilizar la librería descargable de <https://github.com/akafugu/twidisplay>. Esta misma librería es la que también se necesita para poder utilizar otro producto de Akafugu, el "**TWIDisplay 8-digit LCD**", el cual es una pantalla de 8 dígitos 14-segmentos (controlable también vía I²C con los mismos cables).

Otro módulo con backpack I²C y con 4 displays 7-segmentos (concretamente, los también distribuidos por Adafruit de 0,56 pulgadas de altura) es el producto **nº 878** de Adafruit (hay otros similares con diferentes colores: **nº 879**, **nº 880**, **nº 881** y **nº 1002**). Su pin etiquetado como "CLK" se ha de conectar al pin SCL de la placa Arduino (en la UNO es el pin de entrada analógico nº 5) y el etiquetado como "DAT" al pin SDA de la placa (en la UNO es el pin de entrada analógico nº 4); además se ha de conectar su pin "GND" a tierra y su pin de alimentación, etiquetado como "VCC+",

a 5V. Para trabajar con él cómodamente debemos descargar (e instalar) la librería "Adafruit LED Backpack" de <https://github.com/adafruit/Adafruit-LED-Backpack-Library> y la librería "Adafruit GFX"; gracias a ellas mostrar un número en los displays será tan fácil como usar las funciones `print(número)` o `writeDigitNum(posición,cifra)` –para establecer en la memoria del módulo el valor numérico indicado– seguidas siempre de `writeDisplay()` –para realizar el pintado efectivo de ese valor en los displays–, entre otras.

Adafruit también distribuye otros módulos con backpack I²C (y programables mediante las mismas librerías) que asimismo incorporan 4 displays 7-segmentos pero que, en este caso, son de 1,2 pulgadas de alto (de hecho, también distribuidos por Adafruit); se trata de los productos nº 1268, nº 1269 y nº 1270. Por otro lado, también nos pueden interesar módulos de Adafruit con backpack I²C pero ahora con 4 displays 14-segmentos (de 0,54 pulgadas de altura); son los productos: nº 1911, nº 1912, nº 2157, nº 2158, nº 2159 y nº 2160. Para más información, consultar <https://learn.adafruit.com/adafruit-led-backpack>

Por otro lado, también podemos encontrarnos con módulos de 4 displays 7-segmentos que incorporan un backpack de tipo TTL-Serie. Un ejemplo es el llamado "7-segment serial display" de Sparkfun (productos nº 11440 –verde–, nº 11441 –rojo–, nº 11442 –azul–, nº 11443 –amarillo– o nº 11629 –blanco–), el cual incorpora un chip ATmega368 especialmente programado para comunicarse con cualquier receptor/transmisor de tipo SPI o TTL-UART, como los de las placas Arduino (aunque también podría ser reprogramado vía FTDI para convertir el módulo en un componente autónomo).

Para ello, se ha de conectar el pin del módulo etiquetado como "RX" al pin TX de la placa, además de conectarlo a tierra y a la alimentación (de entre 2,6V y 5,5V). Si se desea, podemos hacer que el módulo utilice el protocolo SPI en vez del canal serie para comunicarse: en ese caso debemos conectar sus pines etiquetados como "MOSI", "SCK" y "CSN" a los pines "MOSI", "SCK" y "SS" de la placa Arduino (además de conectarlo a tierra y la alimentación). Sea cual sea el método de comunicación, este módulo es controlado mediante comandos hexadecimales enviados vía `Serial.write()` y los números a mostrar son especificados mediante una línea `Serial.print()` por cada segmento individual uno tras otro, de izquierda a derecha.

A continuación se muestra un código de ejemplo que implementa un contador de segundos y minutos usando la comunicación serie (se recomienda consultar el datasheet del producto para conocer la variedad de comandos posibles):

EL MUNDO GENUINO-ARDUINO

Ejemplo 5.8

```
const int txPin = 1; // Pin de salida serie
const int rxPin = 0; // No se usa, pero se ha de poner
void setup() {
    pinMode(txPin, OUTPUT);
/*Inicializo la comunicación con el display. La velocidad en bits/s se puede cambiar de forma permanente (hasta nuevo cambio) si se le envía el comando 0x7F seguido del código correspondiente al valor deseado. Para más información, consultar el datasheet.*/
    Serial.begin(9600);
    //Comando que pone los cuatro 7-segmentos en blanco. Cada "x" representa un segmento
    Serial.print("xxxx");
    //El comando 0x74 indica que se configura el brillo
    Serial.write(0x7A);
    //Concretamente, establece el brillo bastante bajo (valores posibles son de 0x00 hasta 0xFE)
    Serial.write(0x0F);
    //El comando 0x77 indica que se configurarán las comas decimales
    Serial.write(0x77);
    /*Concretamente, las apaga todas. Si se quiere mostrar la coma decimal, poner: 0x10, pero hay más posibilidades */
    Serial.write(0x0F);
    //Comando que reinicia el display
    Serial.write(0x76);
}
void loop() {
    /*decenaminutos=segmento de más a la izquierda
    minutos=segundo segmento de más a la izquierda
    decenasegundos= tercer segmento de más a la izquierda
    segundos = cuarto segmento de más a la izquierda */
    byte segundos, minutos, decenasegundos, decenaminutos;
    segundos++;
    if(segundos > 9) { segundos = 0; decenasegundos++; }
    if(decenasegundos > 5) { decenasegundos = 0; minutos++; }
    if(minutos > 9) { minutos = 0; deciminutos++; }
    if(decenaminutos > 9) {decenaminutos = 0;Serial.print("xxxx"); }
    if(decenaminutos > 0) {
        Serial.print(decenaminutos); Serial.print(minutos);
        Serial.print(decenasegundos); Serial.print(segundos);
    } else if(minutos > 0) {
        Serial.print("x");           Serial.print(minutos);
        Serial.print(decenasegundos); Serial.print(segundos);
    } else if(decenasegundos > 0) {
        Serial.print("x");           Serial.print("x");
        Serial.print(decenasegundos); Serial.print(segundos);
    } else {
```

```

    Serial.print("x");
    Serial.print("x");
}
delay(1000);
}

```

Si usamos el módulo anterior y necesitamos transformar un número de varias cifras en dígitos individuales para poderlos manipular mejor (por ejemplo, para mostrar un valor obtenido de un sensor, o el número de pulsaciones realizadas sobre un botón, o una simple cuenta atrás que finalice en 0), una buena idea es convertir ese número en un objeto String, para así utilizar funciones como *micadena.length()* (para saber de cuántas cifras es ese número) y sobre todo *micadena.charAt()* (para seleccionar un dígito individual concreto de entre los cuatro posibles, y entonces imprimirla con *Serial.print()*). Se deja como ejercicio.

Matrices de LEDs

Lo bueno de la librería "Adafruit LED Backpack" es que la podemos utilizar sin ningún cambio en otro tipo de displays ofrecidos por Adafruit formados por múltiples LEDs: los módulos de matrices de LEDs 8x8 tales como el producto **nº 870** (y otros similares de diferentes colores, incluyendo la matriz bicolor **nº 902**). Esto es debido a que todos estos productos incorporan el mismo chip controlador HT16K33. Igualmente, requieren cuatro cables para conectarse a una placa Arduino: alimentación (5V), tierra, línea SDA y línea SCL. Para aprender a programarlos se puede consultar el código de ejemplo llamado "matrix88" (o bien el llamado "bimatrix88", para la matriz bicolor) que viene junto con la librería "Adafruit LED Backpack". En estos códigos además se muestra el uso de las funciones de la librería "Adafruit GFX" que permiten dibujar figuras, líneas, etc.

Otros módulos similares al anterior son los productos **nº 759** y **nº 760** de Sparkfun. Ambos incorporan también una matriz de 8x8 LEDs pero, a diferencia del modelo de Adafruit, los de Sparkfun se comunican vía SPI. En el producto **nº 759** la matriz de bicolor (rojo-verde) y en el producto **nº 760** la matriz es de tipo RGB, (permitiendo mostrar hasta siete colores diferentes). No requieren ninguna librería en particular para ser controlados desde una placa Arduino: tan solo el uso de la librería SPI oficial de Arduino. En la página del producto **nº 760** se pueden obtener códigos de ejemplo.

Los módulos anteriores, ya sean los de Sparkfun o el de Adafruit, disponen de la característica de poderse conectar unos a otros en "cascada", de forma que se puedan formar conjuntos de matrices de LEDs de grandes dimensiones.

EL MUNDO GENUINO-ARDUINO

No obstante, si se quieren manejar matrices de LEDs grandes, una solución realmente sencilla es utilizar matrices multicolores de elevadas dimensiones ya de fábrica. Un ejemplo es la "**Dot Matrix Display**" de 32x16 (512) LEDs de Freetronics. Su conexión a Arduino es tremadamente fácil mediante el acoplamiento de una plaquita breakout (incluida en el producto) a los pines digitales nº 6, 7, 8, 9, 10, 11, 12 y 13 de la placa Arduino. A esa plaquita breakout se ha de enchufar un cable de cinta de 22cm (incluido en el producto) que va a parar en el otro extremo al conector pertinente de la pantalla (es el situado más a la izquierda: el conector de la derecha está pensado para conectarlo a otra pantalla de este tipo y hacer así un panel en cadena). Una vez alimentada cada pantalla (mediante su correspondiente conector presente en su dorso) con una fuente externa de 5V y 2,8A, ya está. Para poder controlar esta pantalla desde Arduino, hemos de utilizar una librería propia, la "DMD": <https://github.com/freetronics/DMD>. Esta librería permite la escritura de caracteres y cadenas en diferentes fuentes, así como el pintado de píxeles individuales y el dibujado de líneas y figuras geométricas. En la web del producto se pueden descargar claros códigos de ejemplo, así como la guía de instalación detallada.

Otras pantallas "gigantes" son el producto **nº 420** (16x32 píxeles, 512 LEDs RGB) y el **nº 607** (32x32 píxeles, 1024 LEDs RGB) de Adafruit, muy parecidas a la pantalla de Freetronics. Desgraciadamente, no vienen acompañadas de ninguna plaquita breakout que facilite la conexión del cable de cinta a los pines-hembra de la placa Arduino, así que el cableado es algo engoroso (aunque está bien explicado en la página web del producto). Ambas pantallas se programan mediante la librería "Adafruit RGB Matrix Panel", descargable de aquí: <https://github.com/adafruit/RGB-matrix-Panel> (se requiere también la instalación de la librería "Adafruit GFX").

Adafruit también ofrece el producto **nº 555** (16x24 LEDs rojos), que no es más que un acoplamiento de 6 matrices 8x8. Utiliza también cables de cinta, por lo que la conexión a la placa Arduino no es del todo directa. Es programable mediante la librería "HT1632" (llamada así por el chip controlador que incorporan), descargable de aquí: <https://github.com/adafruit/HT1632>, además de la "Adafruit GFX".

Otras pantallas gigantes que utilizan el mismo chip controlador HT1632 son las "24x16 LED Dot Matrix Unit Boards" de Sure Electronics, con código de producto **DE-DP11111** si los 384 LEDs son de 3mm y de color verde, con código **DE-DP11112** si son de 3mm y de color rojo, con código **DE-DP11211** si son de 5mm y de color verde o con código **DE-DP11212** si son de 5mm y de color rojo. Para programar estas pantallas de SureElectronics, podemos utilizar la librería "HT1632" disponible en <https://github.com/milesburton/HT1632>.

Otro shield es el "**LED Matrix Shield**" de Hobbytronics, que ofrece una matriz 8x8 de color rojo; para programarla no se requiere ninguna librería en especial (en la página del producto se pueden consultar diversos códigos de ejemplo).

Otro shield es el "**Colors Shield**" de IteadStudio, que ofrece un zócalo para poder colocar una matriz 8x8 RGB, la cual no necesita de ninguna librería especial para ser programada (en la página del producto se pueden consultar diversos códigos de ejemplo), aunque existe una librería no oficial que facilita su uso y que puede ser descargada de aquí: <https://github.com/linromatic/Colorduino>.

Incluso, si no queremos usar el combo Arduino+shield, podemos utilizar placas propiamente dichas (en sustitución de la mismísima placa Arduino) especialmente diseñadas para alojar una matriz 8x8 RGB. Estas placas son compatibles con el entorno de programación oficial de Arduino y se pueden programar como si fueran una placa UNO estándar, ya que incorporan el microcontrolador ATmega328P con el bootloader de Arduino y un conector FTDI. Un ejemplo es la placa **Colorduino** de IteadStudio, funcionalmente idéntica a su shield "Colors Shield" pero en forma de placa. Otro ejemplo es la placa **Rainbowduino** de Seeedstudio, que utiliza la librería "Rainbowduino Library" (descargable de la web del producto) para la gestión de los LEDs.

USO DE LA MEMORIA EEPROM

En una memoria EEPROM los datos siempre se guardan en bloques dentro de un número determinado de "celdas", las cuales tienen el tamaño de 1 byte. Así, si el dato (que puede ser de cualquier tipo) está formado por un conjunto de bytes, se almacenará en celdas (bytes) adyacentes.

La librería oficial EEPROM consta de las siguientes funciones de escritura...:

EEPROM.write(): escribe un byte en una celda especificada de la EEPROM. Tiene dos parámetros: el primero –de tipo *int*– es el número de celda donde se escribirá el byte (la numeración empieza por cero; en el modelo UNO –por nombrar uno– existen 1024 celdas, así que sus valores posibles van desde 0 –primera celda– a 1023 –última celda–); el segundo es el valor a escribir (es de tipo *byte*; por tanto, sus valores posibles solamente pueden ser números enteros entre 0 y 255). No tiene valor de retorno.

EEPROM.update(): función idéntica a *EEPROM.write()* en forma (admite los mismos dos parámetros) y fondo (sirve para escribir un byte en una celda

EL MUNDO GENUINO-ARDUINO

específica de la EEPROM) excepto en el hecho de que, antes de realizar cualquier escritura, primero comprueba si el valor a grabar es el mismo que el pudiera existir previamente en la celda a sobrescribir; en el caso de que ambos valores sean el mismo, se ahorrará de hacer la escritura; si no es así, actuará entonces igual que *EEPROM.write()*. La utilidad de esta función radica en el gran ahorro de escrituras que produce (si los cambios en los valores guardados no son frecuentes). Este ahorro es importante por dos motivos: porque una escritura de la EEPROM tarda 3,3ms en completarse (mucho más tiempo de lo que tarda la simple lectura y comparación del valor ya guardado con el candidato a ser escrito), y sobre todo, porque la memoria EEPROM tiene una vida media de 100000 escrituras/borrados, más allá de las cuales no se garantiza la integridad de los datos allí almacenados.

EEPROM.put(): escribe un dato que ocupe más de un byte de memoria (es decir, todos los de tipo *int*, *word*, *short*, *long*, *unsigned long*, *float* e incluso arrays y *struct*) a lo largo de las celdas que sean necesarias. Tiene dos parámetros: el primero –de tipo *int*– es el número de la primera celda a partir de la cual se escribirá ese dato; el segundo es el valor concreto a escribir. Su valor de retorno no nos será relevante.

... y de las siguientes funciones de lectura:

EEPROM.read(): lee un byte de la celda especificada. Tiene un parámetro –de tipo *int*–: el número de celda de donde se lee el byte (número que puede estar entre 0 y 1023, recordemos). Su valor de retorno es el valor del byte leído. Las celdas que no han sido escritas previamente guardan un valor de 255.

EEPROM.get(): obtiene el valor de un dato previamente guardado mediante *EEPROM.put()*. Tiene dos parámetros: el primero –de tipo *int*– es el número de la primera celda de la EEPROM a partir de la cual se leerá el valor; el segundo contendrá precisamente ese valor recién leído; normalmente este segundo parámetro viene indicado en forma de una variable (declarada del mismo tipo que el del valor a obtener) para poder así trabajar a posteriori con el valor albergado en ella. Su valor de retorno no nos será relevante.

Un código autoexplicativo de las funciones *EEPROM.write()* y *EEPROM.read()* es el siguiente:

Ejemplo 5.9

```
#include <EEPROM.h>
int i,celda = 0;
byte valor;
void setup()
{
    Serial.begin(9600);
/*Escribo los primeros 512 bytes de la memoria EEPROM con valores que coinciden con su índice.
Observar cómo cuando se quiere escribir un valor mayor de 255, se empieza de 0 otra vez.*/
    for (i = 0; i < 512; i++){
        EEPROM.write(i, i);
    }
}
void loop()
{
    //Leo el byte ubicado en una celda
    valor = EEPROM.read(celda);
    //Muestro el nº de la celda y el valor acabado de leer
    Serial.print(celda); Serial.print("\t");
    Serial.print(valor); Serial.println();
    //Me muevo a la celda siguiente
    celda = celda + 1;
// Si se llega a la última celda que contiene valores escritos en el setup(), se vuelve otra vez al principio
    if (celda == 512){
        celda = 0;
    }
    delay(50);
}
```

Un código autoexplicativo de *EEPROM.put()* y *EEPROM.get()* es el siguiente:

Ejemplo 5.10

```
#include <EEPROM.h>
int celda;
float numEscrito=123.45;
float numLeido;
struct unaEstruct{ float campo1; word campo2; char nombre[10]; };
unaEstruct datoEscrito = { 3.14,65,"Funciona" };
unaEstruct datoLeido;
void setup(){
    Serial.begin(9600);
/*Se proponen dos ejemplos de escritura: la de un valor decimal (que podría representar la
versión concreta de nuestro firmware, por ejemplo) y la de una estructura (que podría
representar nuestra información de contacto, legal, etc.) */
```

EL MUNDO GENUINO-ARDUINO

```
//Concretamente, primero escribo -a partir de la cuarta celda (nº 3)- un valor decimal  
celda=3;  
EEPROM.put(celda,numEscrito);  
//Avanzo hasta la primera celda libre justo después de ese dato decimal...  
celda = celda + sizeof(float);  
//...y ahora escribo a partir de allí una estructura  
EEPROM.put(celda,datoEscrito);  
//Ahora obtengo el valor decimal guardado en las líneas anteriores (y lo muestro)  
celda=3;  
EEPROM.get(celda,numLeido);  
Serial.println(numLeido); //Si no se obtiene un número decimal válido, se mostrará 'ovf, nan'  
//Y finalmente, obtengo y muestro los campos de la estructura guardada en líneas anteriores  
celda = celda + sizeof(float);  
EEPROM.get(celda,datoLeido);  
Serial.println(datoLeido.campo1);  
Serial.println(datoLeido.campo2);  
Serial.println(datoLeido.nombre);  
}  
void loop(){}
```

Por otro lado, también es posible utilizar el array global predefinido **EEPROM[numCelda]** para leer o escribir un byte directamente en la celda cuyo número se indique como índice (de forma similar a como lo harían las funciones *EEPROM.read()* y *EEPROM.write()*, respectivamente). Un código autoexplicativo de esta opción es el siguiente:

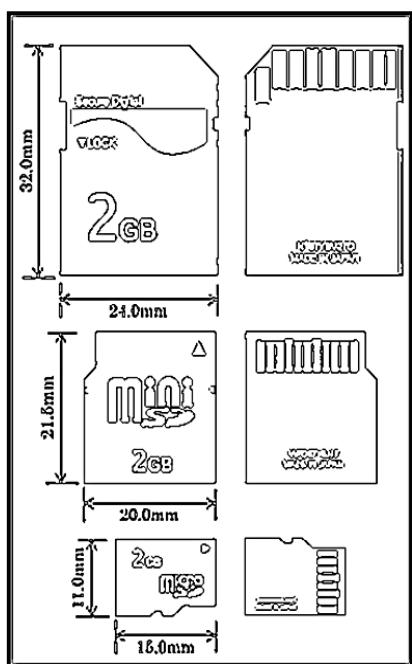
Ejemplo 5.11

```
#include <EEPROM.h>  
void setup(){  
    byte valor;  
    Serial.begin(9600);  
    //Leo primero el valor de la primera celda...  
    valor = EEPROM[ 0 ];  
    //...y lo escribo en la segunda  
    EEPROM[ 1 ] = valor;  
    //Comparo ambos valores. Si son iguales (que debería) se muestra un mensaje por canal serie  
    if( EEPROM[ 0 ] == EEPROM[ 1 ]){  
        Serial.print("Las dos celdas tienen el valor ");  
        Serial.println(valor);  
    }  
}  
void loop(){}
```

Aunque ya sabemos que la memoria EEPROM de la placa Arduino UNO contiene 1024 celdas (siempre de un byte cada una), esto puede ser diferente en otros modelos de placas Arduino. Para conocer en cada caso el número concreto de celdas existentes en la memoria EEPROM de la placa que estemos utilizando, podemos utilizar la función **EEPROM.length()**, la cual tiene como valor de retorno precisamente ese número.

USO DE TARJETAS SD

Características de las tarjetas SD



Secure Digital (SD) es una especificación internacional que define el formato (tanto físico como eléctrico y electrónico) de un determinado tipo de tarjeta generalmente empleado para albergar memoria Flash (por tanto, no volátil). Esta especificación está mantenida por la SD Card Association (<https://www.sdcard.org>), entidad que engloba muchos fabricantes de hardware. Las tarjetas SD originalmente fueron diseñadas para ser usadas como almacenamiento suplementario en pequeños dispositivos portátiles (teléfonos móviles, cámaras digitales, etc.) pero actualmente se emplean en muchos otros aparatos porque ofrecen varias características convenientes, como son su reducido tamaño, su gran durabilidad y retrocompatibilidad, su (relativa) elevada capacidad de almacenamiento y su precio bastante asequible.

El estándar SD incluye 4 "familias" de tarjetas que se diferencian entre sí básicamente por la máxima capacidad de almacenamiento que son capaces de ofrecer. Estas familias son (por orden cronológico de aparición): las tarjetas originales (SDSC –Standard capacity–, que permiten almacenar hasta 4GBytes); las tarjetas de alta capacidad (SDHC –High capacity–, que permiten almacenar hasta 32GBytes); las tarjetas de capacidad extendida (SDXC –Extended capacity–, que permiten almacenar hasta 2048GBytes); y las tarjetas que combinan almacenamiento de datos con funciones de entrada/salida (SDIO). Hay que tener en cuenta que los dispositivos que alojan las tarjetas SD siempre son compatibles "hacia atrás" (es decir: si por ejemplo

EL MUNDO GENUINO-ARDUINO

el dispositivo es compatible con SDXC, también lo será con SDHC y SDSC) pero esto no ocurre al revés: una tarjeta SDXC (por ejemplo) no puede ser utilizada dentro de dispositivos que no sean explícitamente compatibles con esa familia en concreto.

Las formas físicas con las que se comercializan las cuatro familias, tal como se puede ver en la ilustración lateral, son tres: "SD", "miniSD" y "microSD" (también llamadas "TF"). En general, salvo alguna excepción aislada, existen combinaciones de todas las cuatro familias con las tres formas (aunque en la práctica, las tarjetas "miniSD" prácticamente han desaparecido a favor de las "microSD"), de manera que podemos hablar de tarjetas "microSDHC", "miniSDSC", etc. Por fortuna, la disparidad existente de formas no es un gran inconveniente porque en el mercado podemos encontrar adaptadores físicos que permiten el uso de tarjetas pequeñas (generalmente, "microSD") dentro de zócalos grandes (típicamente, "SD") sin problemas. Por ejemplo, el producto **nº 102** de Adafruit consiste en una tarjeta "microSDHC" de 4GBytes acompañada de un adaptador a "SD"; Sparkfun también distribuye, como producto **nº 12998**, una tarjeta "microSDHC" (en este caso de 8GBytes) acompañada de un adaptador a "SD".

NOTA: A no ser que se especifique lo contrario, a partir de ahora por "tarjeta SD" se entenderá indistintamente cualquier tarjeta que tenga tanto la forma "SD" propiamente dicha como la forma "microSD".

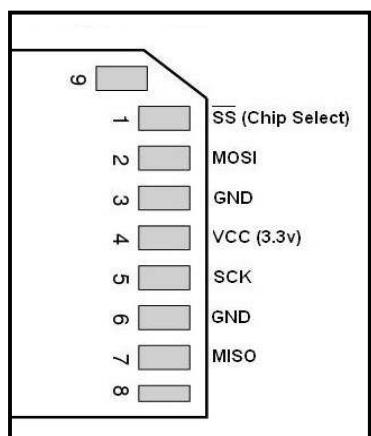
Otro aspecto de las tarjetas SD que conviene conocer es la velocidad mínima de transferencia de datos garantizada, la cual viene definida por la "clase" de la tarjeta en cuestión (que puede ser de cualquier familia). Así, encontramos tarjetas de Clase 2 (2MBytes/s mínimos garantizados), de Clase 4 (4MBytes/s), de Clase 6 (6MBytes/s), de Clase 10 (10MBytes/s) o de Clase UHS 3 (30MBytes/s), entre otras. La velocidad máxima, en cambio, puede variar bastante entre distintas tarjetas de una misma clase o familia; de forma general, las de tipo SDSC no suelen sobrepasar los 25MBytes/s mientras que el modo Ultra-High Speed (UHS) –implementado en algunas tarjetas SDHC y en casi todas las SDXC–, permite velocidades teóricas de hasta aproximadamente 100MBytes/s en su versión "I" y de hasta aproximadamente 300MBytes/s en su versión "II" (siempre y cuando el dispositivo donde se aloje la tarjeta en cuestión sea capaz de gestionar estas velocidades también).

Para que se puedan leer y escribir datos en una tarjeta SD, esta ha de haber sido formateada previamente con un determinado sistema de ficheros. El sistema de ficheros más usado en tarjetas SD es uno llamado FAT32 (evolución de otro anterior llamado FAT16) debido a que la gran mayoría de aparatos electrónicos que utilizan este tipo de tarjetas (cámaras, móviles, etc.) son capaces de reconocer este formato sin problemas. De hecho, la mayoría de tarjetas SD que se suelen adquirir en cualquier tienda local de electrónica ya vienen formateadas con el sistema FAT32.

Solo hay una excepción importante: las tarjetas SDXC, que se comercializan preformateadas con el sistema de ficheros privativo y patentado exFAT de Microsoft (por lo que es posible que algunos dispositivos no sean capaces de leer su contenido).

Si necesitamos formatear una tarjeta SD en el sistema FAT32 (bien porque la adquirimos sin formato o bien porque le queremos quitar el que tiene) primero debemos conectarla mediante un lector de tarjetas apropiado a un computador para que este la reconozca como un dispositivo de almacenamiento más. Si dicho computador no incorpora ningún zócalo SD/microSD de fábrica, siempre es posible utilizar un adaptador USB<->SD/microSD, los cuales ofrecen por un lado un zócalo SD/microSD para poder insertar allí la tarjeta pertinente y por otro un conector USB –de tipo A– para enchufarlo a nuestro computador. Un ejemplo de estos adaptadores es el que se distribuye en Sparkfun como producto nº 13004 y también (junto con una tarjeta microSD de 8GBytes y un adaptador SD) como producto nº 11609. Una vez reconocida la tarjeta, el proceso concreto a seguir para formatearla dependerá del sistema operativo instalado en el computador y de las aplicaciones especializadas proporcionadas por este. De todas formas, desde la SD Card Association recomiendan usar la aplicación de formateo desarrollada por ellos mismos (solo disponible, no obstante, para sistemas Windows y OS X) descargable gratuitamente de https://www.sdcard.org/downloads/formatter_4.

Shields y módulos que incorporan zócalos microSD



Las tarjetas SD soportan varios modos de transferencia de datos. Cada uno de estos modos utiliza de forma distinta los pines de la tarjeta; así, podemos encontrarnos con el modo "un-bit" (un protocolo serie específico) o el modo "cuatro-bit" (que soporta transferencias paralelas de cuatro bits), entre otros. Desgraciadamente, todos estos modos están cubiertos por varias patentes y solo se pueden licenciar a través de la SD Card Association pagando una elevada suma de dinero. Afortunadamente, hay un modo que no está afectado por este asunto legal y que, por tanto, aunque sea algo más lento que los otros modos, puede ser utilizado sin problemas (y es

el que se utiliza, de hecho) para poder establecer comunicación vía SPI con las tarjetas SD desde, por ejemplo, una placa Arduino. Este modo (que las tarjetas SD implementan siempre) se llama "modo SPI/MMC" porque emula el comportamiento de las antiguas tarjetas MultiMediaCard. Más en concreto, el diagrama lateral (que

EL MUNDO GENUINO-ARDUINO

representa el dorso de un tarjeta SD –las tarjetas microSD son idénticas excepto en que carecen del pin nº 9– muestra la función que tiene cada pin: como se puede observar, algunos de los pines de la tarjeta son usados como líneas SPI (MOSI, MISO, SCK y SS) y otros se reservan para conexiones a alimentación, tierra o nada.

Desgraciadamente, no es posible conectar una tarjeta SD directamente a una placa Arduino porque las líneas de cualquier tarjeta SD (tanto la de alimentación como todas las de datos) funcionan a 3,3V. Esto obliga a que tengamos que utilizar siempre un convertidor de nivel como intermediario entre ambos dispositivos. Otra solución –mucho más cómoda– es optar por alguno de los shields que hay disponibles en el mercado con el zócalo SD/microSD y la circuitería pertinente ya integrada. Un ejemplo de ello es el producto nº **12761** de Sparkfun (ofrece un zócalo de tipo microSD y además una zona de prototipado; se puede programar mediante librería oficial de Arduino, aunque utiliza el pin nº 8 como SS). Otro shield interesante (que utiliza el pin nº 4 como SS y que también es programable mediante la librería oficial de Arduino) es el "**SD card shield**" de Seeedstudio, el cual admite tanto tarjetas SD como microSD. Shield similares al anterior son el "**Stackable SD card shield**" de Iteadstudio o el "**SD Card Shield**" de Makerstudio.

También es habitual encontrarnos con shields que, además de ofrecer un zócalo SD/microSD y, eventualmente, una zona de prototipado, también ofrecen un reloj de tiempo real (un RTC); esto es así porque en muchas ocasiones nos interesará guardar en la tarjeta SD datos (obtenidos generalmente de sensores) junto con la fecha y hora concreta a la que fueron tomados. Ejemplos de estos shields son el "**Memoire shield**" de Snootlab (cuyo zócalo SD utiliza el pin nº 10 como SS y es programable mediante la librería oficial de Arduino; incorpora por otro lado el reloj DS1307 de Maxim, programable mediante cualquier librería compatible como <https://github.com/olikraus/ds1307new>) o el producto nº **1141** de Adafruit (cuyo zócalo SD utiliza también el pin nº 10 como SS, también es programmable mediante la librería oficial de Arduino y también incorpora un reloj RTC DS1307 –esta vez programable mediante la librería <https://github.com/adafruit/RTClib> y conectado con la placa Arduino a través de los pines I²C– alimentado por una pila botón CR1220).

También podemos utilizar en nuestros proyectos módulos independientes que no sean shields. Todos ellos se programan igualmente con la misma librería SD oficial de Arduino. Un ejemplo puede ser el producto nº **254** de Adafruit, el cual ofrece los siguientes pines: el "5V" (a conectar directamente al pin 5V de la placa Arduino), el "GND" (a conectar a algún pin GND) y los necesarios para la comunicación SPI: el "CLK" (a conectar al pin nº 13 de Arduino), el "DO" (–de "Data Output"– a conectar al pin nº 12), el "DI" (–de "Data Input"– a conectar al pin nº 11) y

el "CS" (a conectar a cualquier pin digital, como por ejemplo el nº 10). Otros módulos muy parecidos al anterior son los "**SD Module**" y "**MicroSD Card Module**" de DFRobot o los "**SD/MMC Card Adapter**" y "**MicroSD Card Adapter**" de Gravitech.

Por otro lado, un módulo algo diferente pero digno de destacar es el OpenLog de Sparkfun (producto **nº 9530**), consistente en una placa breakout con un zócalo microSD que se comunica a través del canal serie (pines RX y TX) con la placa Arduino. Su funcionalidad más evidente es grabar los datos recibidos por ese canal serie, pero lo original es que a través de ese mismo canal (es decir, usando simplemente *Serial.write()*) es posible enviar una serie de comandos propios –inspirados en los comandos más habituales del terminal de Linux– que permiten crear ficheros, borrarlos, listarlos, etc. Otro módulo similar (pero con otro conjunto de comandos propios, esta vez de tipo hexadecimal) es el "**SmartDRIVE**" de Vizic Technologies.

La librería SD

Antes de conocer las instrucciones que provee la librería oficial SD, hay que tener en cuenta una serie de consideraciones que pasamos a comentar.

1. La librería oficial SD solamente permite nombre de ficheros con una longitud máxima de 8 caracteres más 3 caracteres para su extensión. La razón principal de esta limitación es el ahorro que supone en el consumo de memoria SRAM realizado por el sketch en cuestión. Si quisieramos, no obstante, reconocer nombres más extensos, tenemos una solución: utilizar, en vez de la librería oficial, otra librería llamada "SdFat" (<https://github.com/greiman/SdFat>), la cual aporta varias mejoras más no presentes en la librería oficial como, por ejemplo, la capacidad de gestionar varias tarjetas SD a la vez, entre otras.

2. El sistema de ficheros FAT32 (y FAT16) no distingue mayúsculas de minúsculas en los nombres de ficheros y carpetas; es decir: es lo mismo "mifichero.txt" que "mifichero.TXT" que "mifichero.tXt". A esto se le llama "nombres de ficheros *case-insensitive*".

3. Los nombres de ficheros pasados como parámetros en las distintas funciones de la librería oficial SD pueden ser en realidad rutas completas (es decir, pueden incluir el nombre de las distintas carpetas a las que se ha de acceder una tras otra hasta llegar finalmente al fichero en cuestión). Esta ruta se escribe separando cada subcarpeta intermedia con el signo de la barra "/", por ejemplo así: "carpeta/fichero.txt" (al igual que ocurre de hecho en los sistemas Linux y OS X).

EL MUNDO GENUINO-ARDUINO

4. Por defecto la carpeta de trabajo es la carpeta "raíz" de la tarjeta (la carpeta llamada ""). Esto hace que especificar el nombre de un fichero "a secas" sea equivalente a especificar ese nombre precedido de una barra. Es decir, "/fichero.txt" equivale a "fichero.txt".

5. La librería oficial SD asume por defecto que la comunicación entre el microcontrolador de la placa Arduino UNO y la tarjeta SD (esté montada esta sobre un shield o módulo cualquiera) se establece mediante el protocolo SPI (es decir, a través de los pines digitales nº 11, 12 y 13) usando como pin SS el nº 10. No obstante, si la tarjeta SD elegida utilizará otro pin para la funcionalidad SS (por el diseño hardware concreto del shield/módulo donde está ubicada) es posible configurar la librería para reconocer dicho otro pin como pin SS en vez del nº 10. Por ejemplo, en el caso de la placa Arduino Ethernet o el shield Arduino Ethernet, el pin SS utilizado por la tarjeta SD por defecto es el número 4, ya que el número 10 se reserva para la comunicación interna con el chip Ethernet. De todas maneras, es muy importante tener en cuenta que, aunque no se utilice el pin 10 como SS, este deberá ser configurado en nuestros sketches siempre explícitamente como salida digital (OUTPUT), o si no la librería SD no funcionará.

La primera instrucción de la librería SD que debemos ejecutar antes de cualquier otra (normalmente dentro de la función *setup()*) es:

SD.begin(): inicializa la librería y la tarjeta SD. Opcionalmente, admite un parámetro para especificar el número de pin de la placa Arduino que la librería utilizará como canal SS en la comunicación SPI con la tarjeta SD. Si este parámetro no se especifica, por defecto la librería siempre interpretará como SS el pin número 10, pero, si, por ejemplo, estuviéramos usando la placa Arduino Ethernet (o el shield con el mismo nombre), el pin que deberíamos utilizar como pin SS es el número 4, por lo que entonces deberíamos invocar esta función así: SD.begin(4);. En todo caso, hay que tener en cuenta que aunque la librería SD utilice un pin diferente del número 10, ese pin número 10 siempre ha de configurarse como salida digital (OUTPUT) para que todo funcione correctamente. El valor de retorno de esta función es de tipo booleano: valdrá 1 si la orden se ejecuta correctamente y 0 si ocurre algún error (como por ejemplo, que no se haya detectado ninguna tarjeta en el módulo).

También tenemos estas otras funciones de la librería SD:

SD.exists(): comprueba si un fichero (o carpeta) pasado por parámetro existe en la tarjeta SD. Como único parámetro tiene el nombre o ruta del fichero (o

carpeta) del cual se quiere comprobar su existencia. Su valor de retorno es de tipo booleano: valdrá 1 si existe, 0 si no.

SD.mkdir(): crea una carpeta en la tarjeta SD. Como único parámetro tiene el nombre o ruta de la carpeta que se quiere crear. Si la carpeta ya existiera, esta función no hace nada. Si se especifican carpetas intermedias que no existen, también se crean (por ejemplo, si escribimos SD.mkdir("a/b/c"); se crearán la carpeta "a", y dentro de esta, "b", y dentro de esta, "c". Su valor de retorno es de tipo booleano: valdrá 1 si la orden se ejecuta correctamente, 0 si ocurre algún error.

SD.rmdir(): elimina una carpeta de la tarjeta SD. Esta carpeta ha de estar vacía para poder ser borrada. Como único parámetro tiene el nombre o ruta de la carpeta que se quiere eliminar. Su valor de retorno es de tipo booleano: valdrá 1 si el borrado se ha realizado correctamente o 0 si ha ocurrido algún error, pero si la carpeta especificada no existiera, el valor de retorno está no determinado.

SD.remove(): elimina un fichero de la tarjeta SD. Como único parámetro tiene el nombre o la ruta del fichero que se quiere eliminar. Su valor de retorno es booleano: valdrá 1 si el borrado se ha realizado correctamente o 0 si ha ocurrido algún error; si el fichero especificado no existiera, el valor de retorno está no determinado.

Y sobre todo:

SD.open(): "abre" un fichero en la tarjeta SD. Abrir un fichero significa poder empezar a leer o bien editar su contenido. En el caso concreto de que se abra para escritura un fichero que no exista, será creado. Es posible tener múltiples ficheros abiertos (y manipularlos) a la vez. La importancia de la función *SD.open()* radica en que devuelve un objeto de tipo *File*, que representa el fichero abierto. La variable que recoja este objeto devuelto (la llamaremos *mifichero*) debe declararse previamente en nuestro sketch como de tipo *File*, así: File mifichero;. La importancia del objeto *File* devuelto por *SD.open()* está en que este objeto contiene a su vez un conjunto de instrucciones que permiten manipular individualmente ese fichero que representa, tal como en seguida veremos. Como primer parámetro *SD.open()* tiene el nombre o ruta del fichero que se quiere abrir. Como segundo parámetro (opcional) tiene el modo en el que se quiere abrir el fichero; puede tener dos valores posibles: o bien la constante FILE_READ (el modo por

EL MUNDO GENUINO-ARDUINO

defecto), que indica que el fichero se abre en solo lectura empezando por su principio, o bien la constante FILE_WRITE, que indica que el fichero se abre para leer y también escribir en él empezando por su final (para así añadir contenido nuevo sin sobrescribir el existente).

Una vez creado el objeto de tipo *File* mediante la función *SD.open()*, a partir de aquí podemos manipular este fichero en particular con diferentes instrucciones propias de ese objeto. Suponiendo que ese objeto de tipo *File* lo llamamos *mifichero*, las funciones que nos permiten escribir datos en una tarjeta son:

mifichero.print(): escribe el dato pasado como parámetro (que puede ser tanto de tipo *char* o cadena como también entero) dentro del fichero *mifichero*, el cual lógicamente debe haber sido abierto en modo escritura. Si el dato es numérico entero, es tratado como caracteres ASCII (es decir, el número 123 es escrito como tres caracteres, '1','2' y '3'). Opcionalmente, como segundo parámetro se puede especificar (si el valor a enviar es entero) una de las constantes predefinidas BIN, HEX o DEC, para elegir el sistema de numeración empleado para representar ese dato (binario, hexadecimal o decimal –por defecto–, respectivamente). Su valor de retorno es de tipo *byte* e indica el número de bytes escritos; su uso es opcional.

mifichero.println(): hace exactamente lo mismo que *mifichero.print()*, pero además, siempre añade automáticamente al final de los datos escritos dos caracteres: el de retorno de carro (código ASCII nº 13) y el de nueva línea (código ASCII nº 10).

mifichero.write(): escribe el dato que se haya pasado como parámetro dentro del fichero *mifichero* (que debe haber sido abierto en modo escritura). Esta función puede escribirse de dos maneras: si el dato a grabar solo ocupa un byte o si ocupa más. En el primer caso, esta función aceptará un único parámetro que deberá ser de tipo *byte* o *char*. En el segundo caso, esta función puede seguir aceptando un único parámetro, que será una cadena de caracteres, o bien aceptar dos parámetros: el nombre de un array de datos de tipo *byte* o *char* y el número de elementos que se escribirán en él (el cual no tiene por qué coincidir con el número total de elementos del array). Su valor de retorno es de tipo *byte* e indica el número de bytes escritos; su uso es opcional.

Sea como sea, hay que tener en cuenta, no obstante, que las funciones *mifichero.print()*, *mifichero.println()* y *mifichero.write()* no graban físicamente el dato

en la tarjeta: para ello es necesario ejecutar una de las dos funciones siguientes. Es decir, hay que asegurarse siempre de utilizar alguna de estas dos funciones después de utilizar las funciones de escritura de datos para que se guarden realmente en la tarjeta.

mifichero.close(): cierra el fichero *mifichero*. Antes se asegura de escribir físicamente en la tarjeta cualquier dato pendiente de grabar en ese fichero. No tiene ni parámetros ni valor de retorno.

mifichero.flush(): se asegura de escribir físicamente en la tarjeta cualquier dato pendiente de grabar en el fichero *mifichero*. Esto también se realiza automáticamente cada vez que el fichero es cerrado (con *mifichero.close()*). No tiene ni parámetros ni valor de retorno.

Para leer datos de la tarjeta SD, disponemos de las siguientes funciones:

mifichero.available(): comprueba si hay algún byte todavía por leer dentro de *mifichero*. Su valor de retorno (de tipo *int*) es el número de bytes aún disponibles para leer. No tiene parámetros.

mifichero.read(): lee un byte del fichero *mifichero* y avanza al siguiente byte, de manera que la próxima ejecución de *mifichero.read()* lea ese siguiente byte. Su valor de retorno es el valor del byte (o carácter) leído, o -1 si ya no hay más bytes por leer. No tiene parámetros.

mifichero.peek(): lee un byte del fichero *mifichero* sin avanzar al siguiente. Es decir, ejecuciones seguidas de *mifichero.peek()* devolverán siempre el mismo valor, hasta que se ejecute *mifichero.read()*—momento en el que se avanzará hacia el siguiente byte-. Su valor de retorno es el valor del byte (o carácter) leído, o -1 si ya no hay más bytes por leer. No tiene parámetros.

También disponemos de funciones para posicionarse en cualquier byte dentro del interior de un fichero:

mifichero.size(): devuelve el tamaño del fichero *mifichero* en bytes (es un valor de tipo *unsigned long*). No tiene parámetros.

mifichero.position(): devuelve la posición actual del cursor dentro del fichero *mifichero*. Es decir, el lugar dentro de ese fichero a partir del cual se leerá o escribirá el siguiente byte. Esta posición indica en realidad el número de un byte, contando byte a byte desde el principio del fichero. La primera posición es la número 0. Este valor devuelto es de tipo *unsigned long*. No tiene parámetros.

EL MUNDO GENUINO-ARDUINO

mifichero.seek(): cambia la posición del cursor dentro del fichero *mifichero* a la ubicación dada como parámetro. Esta posición indica en realidad el número de un byte, contando byte a byte desde el principio del fichero. Debe valer entre 0 y el tamaño en bytes del fichero (ambos incluidos), y es de tipo *unsigned long*. Su valor de retorno es de tipo booleano: vale 1 si la orden se ejecuta correctamente o 0 si ocurre algún error.

Finalmente, también disponemos de funciones especializadas en trabajar dentro de una carpeta determinada con múltiples ficheros (uno tras otro):

mifichero.isDirectory(): las carpetas (también llamadas a menudo "directorios") son tipos especiales de ficheros. Esta función informa de si *mifichero* es un fichero "estándar" o bien es una carpeta mediante su valor de retorno booleano: si vale 1 *mifichero* es una carpeta y si vale 0, no. No tiene parámetros.

undirectorio.openNextFile(): devuelve un objeto *File* que representa el siguiente fichero encontrado (por orden de creación) existente dentro del directorio *undirectorio*. Si no hay ya ninguno más, devolverá *false*. No tiene parámetros.

mifichero.name(): devuelve el nombre (en formato de 8 caracteres más los 3 de la extensión) del objeto *mifichero*. No tiene ni parámetros ni valor de retorno.

undirectorio.rewindDirectory(): retrocede hasta el primer fichero de la lista de ficheros existentes dentro de la carpeta *undirectorio*. Se suele usar conjuntamente con la instrucción *undirectorio.openNextFile()*. No tiene ni parámetros ni valor de retorno

Una vez explicadas todas las funciones pertenecientes a la librería SD, veamos unos cuantos ejemplos. El siguiente código muestra cómo escribir una frase dentro de un fichero alojado en una tarjeta SD:

Ejemplo 5.12

```
#include <SD.h>
File miarchivo;
void setup(){
/* Establecemos el pin nº 10 como salida digital, o si no la librería SD no funcionará.
Esto es lo que hace la siguiente línea, aunque no la hayamos estudiado todavía.*/
```

```

pinMode(10, OUTPUT);
//Si ha habido un error, se aborta el programa
if (SD.begin(10) == 0) {
    return;
}
/*Se abre el fichero para poder escribir dentro de él (a partir de su última línea existente).
   Si el fichero indicado no existe, lo creará.*/
miarchivo = SD.open("test.txt", FILE_WRITE);
//Si el fichero se abre ok, se escribe una frase y se cierra
if (miarchivo != 0) {
    miarchivo.println("Probando 1, 2, 3.");
    miarchivo.close(); //Imprescindible para que la escritura se realice efectivamente
}
void loop(){}

```

Si en vez de escribir en la tarjeta una frase literal, hubiéramos escrito los datos obtenidos por uno (o más) sensores, ya tendríamos un sketch que nos permitiría guardar de forma permanente esta información y así poder, por ejemplo, realizar un estudio posterior pormenorizado utilizando una hoja de cálculo en un computador. He aquí cómo hacer para leer el dato grabado en el anterior ejemplo:

Ejemplo 5.13

```

#include <SD.h>
File miarchivo;
void setup(){
    Serial.begin(9600);
    pinMode(10, OUTPUT);
    if (SD.begin(10) == 0) {return;}
    miarchivo = SD.open("test.txt");
    if (miarchivo != 0) {
        //Lee del fichero hasta que ya no quede ningún byte por leer
        while (miarchivo.available() > 0) {
            //El byte leído lo muestro en el "Serial monitor"
            Serial.write(miarchivo.read());
        }
        //Siempre es bueno cerrar el fichero una vez listo para ahorrar SRAM
        miarchivo.close();
    }
}
void loop() {}

```

EL MUNDO GENUINO-ARDUINO

El siguiente ejemplo muestra cómo escribir en un fichero sobrescribiendo su contenido anterior. El truco utilizado ha sido borrarlo y volverlo a crear, aunque una manera algo más elegante sería emplear *miarchivo.seek()* para situar el cursor en el byte 0 del archivo:

Ejemplo 5.14

```
#include <SD.h>
File miarchivo;
void setup(){
    pinMode(10, OUTPUT);
    if (SD.begin(10) == 0) {
        return;
    }
    if (SD.exists("test.txt") !=0){
        SD.remove("test.txt");
    }
    miarchivo = SD.open("test.txt", FILE_WRITE);
    if (miarchivo != 0) {
        //miarchivo.seek(0); (Una manera alternativa de sobrescribir)
        miarchivo.println("Probando 1, 2, 3.");
        miarchivo.close();
    }
}
void loop(){}  
}
```

El siguiente código muestra cómo detectar en cada arranque de Arduino si, al querer crear un nuevo fichero, ya hay grabado de antes otro fichero con el mismo nombre. Si ese es el caso, el sketch cambia dinámicamente el nombre del nuevo fichero a crear por otro, para no modificar en absoluto el fichero previamente grabado. Haciendo esto, en cada arranque de Arduino generaremos un fichero diferente, independiente de los generados en anteriores arranques.

Ejemplo 5.15

```
#include <SD.h>
File miarchivo;
void setup(){
    pinMode(10, OUTPUT);
    SD.begin(10);
    //Empiezo con este nombre
    char nombrefichero[]="LOGGER00.CSV";
    /*El bucle sirve para ir probando combinaciones de nombres de ficheros
    hasta que se encuentre una que no está utilizada todavía */
```

```

for (byte i = 0; i < 100; i++) {
    /*El truco para probar combinaciones es cambiar los dos últimos caracteres del nombre del fichero para
    que sean consecutivamente "00","01","02","03"..."10","11","12"...hasta "99" (por tanto, solo
    podemos tener hasta 100 ficheros diferentes). Para conseguir estas combinaciones, utilizamos el
    resultado y el resto de una división de un contador entre diez. A ello se le suma el carácter ASCII '0'
    (equivalente al número 48) para convertir el dígito obtenido en ambas operaciones en su carácter
    ASCII correspondiente (es decir, para convertir por ejemplo el número 6 en el carácter "6", que tiene
    código ASCII número 54) */
    nombrfichero[6] = i/10 + '0';
    nombrfichero[7] = i%10 + '0';
    //Si no existe la combinación actual como nombre de fichero...
    if (!SD.exists(nombrfichero)) {
        //...creo el nuevo fichero con esa combinación como nombre
        miarchivo = SD.open(nombrfichero, FILE_WRITE);
        break; //...y no sigo probando de crear ningún fichero más
    }
}
//En cada reinicio de Arduino, tendré un nuevo fichero con la misma frase
if (miarchivo != 0) {
    miarchivo.println("Probando 1, 2, 3.");
    miarchivo.close();
}
void loop(){}

```

El siguiente código (algo más complejo) muestra la lista de los nombres y tamaños de los ficheros presentes dentro de la carpeta indicada (concretamente la carpeta "raíz"). Se puede ver el uso de una función propia recursiva, la cual muestra los ficheros encontrados tabulados según la subcarpeta a la que pertenezcan.

Ejemplo 5.16

```

#include <SD.h>
File raiz;
File entrada;
int i;
void setup() {
    Serial.begin(9600);
    pinMode(10, OUTPUT);
    if (SD.begin(10) == 0) {return; }
    raiz = SD.open("/");
    printDirectory(root, 0);
}
void loop(){}

```

```
void printDirectory(File dir, int numTabs) {  
    while(true) { //Bucle infinito  
        entrada = raiz.openNextFile();  
        /*No hay más ficheros, así que acabo la ejecución. Si la  
         función se invocó recursivamente, vuelvo a la anterior*/  
        if (entrada == 0) { break; }  
        //Pongo los tabuladores oportunos antes del nombre del fichero  
        for (i=0; i<numTabs; i++) {  
            Serial.print("\t");  
        }  
        //Escribo el nombre del fichero  
        Serial.print(entrada.name());  
        /*Si es una subcarpeta, aumento en uno la tabulación y vuelvo a llamar a la  
         misma función para que haga todo el recorrido de ficheros por esa subcarpeta*/  
        if (entrada.isDirectory() !=0 ) {  
            Serial.println("/");  
            printDirectory(entrada, numTabs+1);  
        //Si no, se muestra al lado del nombre el tamaño del fichero  
        } else {  
            Serial.print("\t\t");  
            Serial.println(entrada.size(), DEC);  
        }  
    }  
}
```

Dentro del conjunto de sketches de ejemplo que vienen "de fábrica" con el IDE Arduino, hay uno en especial que es muy interesante, el "CardInfo". Su código es relativamente complejo y utiliza objetos internos avanzados de la librería que no hemos estudiado, pero su utilidad radica en ejecutarlo para comprobar las características de la tarjeta SD que tengamos conectada en ese momento a la placa Arduino. Concretamente, obtiene el tipo de tarjeta (si es SD o SDHC), el tamaño en KBytes y MBytes de la partición FAT16 o FAT32 detectada, y la lista de ficheros almacenados (mostrando su nombre, fecha de última modificación y tamaño en bytes).

USO DE PUERTOS SERIE SOFTWARE

Lo primero que debemos hacer para poder utilizar una pareja de pines extra como RX y TX mediante la librería oficial SoftwareSerial es declarar una variable global de tipo SoftwareSerial, que representará dentro de nuestro sketch a este nuevo canal serie "virtual". La declaración se ha de realizar usando la siguiente

sintaxis (suponiendo que llamamos *miserie* a dicha variable): *SoftwareSerial miserie(npinrx,npintx);* donde *npinrx* es en realidad un valor numérico que indica el pin de la placa que hará la función RX (es decir, el que recibirá datos del exterior) y *npintx* es en realidad un valor numérico que indica el pin de la placa Arduino que hará la función TX (es decir, el que enviará datos al exterior). Un ejemplo de declaración podría ser, pues: *SoftwareSerial miserie(2,3);*

Una vez creado el canal *miserie* con la línea anterior, lo primero que debemos hacer (normalmente dentro de la función *setup()*) es abrir la conexión mediante la instrucción ***miserie.begin()***, la cual funciona exactamente igual que la ya conocida *Serial.begin()*. A partir de aquí ya podemos utilizar un conjunto de funciones que nos permitirán gestionar ese canal *miserie* en particular. Por ejemplo, podemos hacer uso de ***miserie.available()***, ***miserie.peek()*** y ***miserie.read()*** para recibir datos, y ***miserie.print()***, ***miserie.println()*** y ***miserie.write()*** para enviarlos. Todas ellas son funciones equivalentes a las ya vistas cuando se trató el objeto Serial. Las únicas funciones novedosas que aporta un objeto SoftwareSerial respecto al objeto Serial son las siguientes:

***miserie.listen()*:** en el caso de que exista más de un objeto SoftwareSerial, activa el objeto llamado *miserie* para que pueda recibir datos (o dicho de otra forma, "para ponerlo a la escucha"). Solo un objeto SoftwareSerial puede recibir datos ("escuchar") en un determinado momento: los datos que lleguen al resto de puertos son desechados. Si este comportamiento no es admisible en nuestro proyecto, una alternativa es utilizar una librería llamada "AltSoftSerial" (http://www.pjrc.com/teensy/td_libs_AltSoftSerial.html) que no tiene esta limitación de escuchas y, además, permite el envío y la recepción de datos simultáneamente, algo que SoftwareSerial tampoco es capaz de hacer (pero tiene otras limitaciones, como por ejemplo el no poder utilizar cualquier pin de la placa como RX y TX, sino unos pocos en concreto). Esta función no tiene ni parámetros ni valor de retorno.

***miserie.isListening()*:** comprueba si *miserie* es el puerto serie virtual (de entre los posibles existentes) que está ahora mismo escuchando. Su valor de retorno es booleano: valdrá *true* si está escuchando y *false* si no. No tiene parámetros.

***miserie.overflow()*:** comprueba si la cantidad de datos recibidos ha sobrepasado el límite de almacenamiento del buffer de entrada de *miserie*. El tamaño de los buffers de los objetos SoftwareSerial es de 64 bytes, al igual que el del objeto Serial hardware. Su valor de retorno es booleano: valdrá *true* si se ha superado el límite y *false* si no. No tiene parámetro.

EL MUNDO GENUINO-ARDUINO

El sketch siguiente es un ejemplo sencillo que ilustra el uso de esta librería, donde se va alternando el uso de dos canales serie por software:

Ejemplo 5.17

```
#include <SoftwareSerial.h>
//RX = pin digital 10, TX = pin digital 11
SoftwareSerial canalUno(10, 11);
//RX = pin digital 8, TX = pin digital 9
SoftwareSerial canalDos(8, 9);
void setup(){
    Serial.begin(9600);
    canalUno.begin(9600);
    canalDos.begin(9600);
}
void loop(){
    char dato;
    //Primero se escucha por el puerto Uno
    canalUno.listen();
    Serial.println("Datos recibidos por el puerto Uno:");
    /*Mientras se reciban datos por puerto Uno, se leen byte a byte y
     * se van enviando al puerto serie hardware para verlos por el "Serial monitor" */
    while (canalUno.available() > 0) {
        dato = canalUno.read();
        Serial.write(dato);
    }
    //Línea en blanco para separar los datos recibidos de cada puerto
    Serial.println();
    //Ahora se escucha por el puerto Dos
    canalDos.listen();
    Serial.println("Datos recibidos por el puerto Dos:");
    while (canalDos.available() > 0) {
        dato = canalDos.read();
        Serial.write(dato);
    }
    Serial.println();
}
```

El siguiente ejemplo lo podríamos realizar mediante los canales serie hardware de la placa Arduino UNO (recordemos, el pin nº 0 es el RX y el nº 1 es el TX) pero lo veremos utilizando la librería SoftwareSerial para demostrar que los canales "emulados" funcionan exactamente igual. Para probarlo deberemos tener dos placas Arduino UNO y comunicarlas mutuamente mediante dos cables; concretamente, uno de esos cables deberá estar conectado al pin-hembra nº 2 de una placa (el cual hará

de receptor –RX₁–) y al pin-hembra nº 3 de la otra (el cual hará de transmisor –TX₂–), y el otro cable deberá estar conectado al pin-hembra nº 3 de la primera (el cual hará de transmisor –TX₁–) y al pin-hembra nº 2 de la segunda (el cual hará de receptor –RX₂–). Además, para que la comunicación serie funcione, es muy importante que las tierras de ambas placas estén compartidas (de forma que puedan tener una referencia de niveles de tensión común). Para ello deberemos unir entre sí un pin-hembra "GND" de sendas placas (mediante otro cable). Así pues, en realidad el montaje final necesitará realmente tres cables (los cuales, desgraciadamente, solo podrán tener una longitud de como mucho un metro aproximadamente debido a la inevitable resistencia eléctrica que tienen y que "diluye" la señal eléctrica TTL).

El montaje descrito en el párrafo anterior permite que los dos cables que forman el canal serie sean capaces en conjunto de interconectar bidireccionalmente ambas placas. No obstante, en el código de ejemplo siguiente tan solo se realiza la transferencia de datos en un solo sentido. Concretamente, dicho código solo transmite la información numérica generada por una placa (que podría haber sido obtenida por ejemplo de algún sensor, aunque en el código de ejemplo está escrita explícitamente para simplificar la demostración) a la otra (que podría servir para activar por ejemplo algún actuador, aunque en el código de ejemplo simplemente se muestra por el "Serial monitor") y ya está. En este caso, por tanto, con un cable entre el pin TX de una placa y el RX de la otra y otro uniendo sus respectivas tierras ya bastaría.

Ejemplo 5.18

```
Código del primer Arduino
#include <SoftwareSerial.h>
SoftwareSerial emisor(2,3);
int i = 0;
void setup() {
    emisor.begin(9600);
}
void loop(){
    emisor.print(i);
    i++;
    delay(500);
}
```

```
Código del segundo Arduino
#include <SoftwareSerial.h>
SoftwareSerial receptor(2,3);
void setup() {
    receptor.begin(9600);
```

EL MUNDO GENUINO-ARDUINO

```
    Serial.begin(9600);
}
void loop() {
    if (receptor.available() > 0) {
        Serial.print(receptor.parseInt());
    }
}
```

De todas formas, una alternativa para tener un control más completo y flexible de los datos transferidos entre dos Arduinos (ya sea utilizando los pines RX y TX hardware, los pines SoftwareSerial o incluso mediante el protocolo I²C!), es utilizar una librería muy práctica que nos permite manejar la información directamente en formato binario, ofreciendo así una forma más compacta –y por tanto, eficiente y rápida– de transferir datos que si usamos caracteres ASCII: la llamada "Arduino Easy-Transfer", descargable de <https://github.com/madsci1016/Arduino-EasyTransfer>. Recomiendo consultar los sketches de ejemplo que vienen incluidos junto con ella para conocer su uso.

Es importante tener en cuenta, finalmente, que los puertos serie por software consumen tiempo y además necesitan recursos extra (básicamente espacio en memoria SRAM y disponibilidad para operaciones en el microcontrolador): un puerto serie por software debe hacer todo lo que hace un puerto serie por hardware pero usando para ello el mismo microcontrolador que intenta ejecutar a la vez nuestro sketch concreto. Por ejemplo, si un nuevo carácter llega a un puerto serie por software, el microcontrolador deberá interrumpir lo que esté haciendo, recibir y procesar ese nuevo carácter (acción que lleva su tiempo –y, lógicamente, cuantos más caracteres recibidos, mayor parón–) y solo cuando esta entrada de datos haya concluido, reanudar la tarea pendiente. La moraleja de esto es que si es necesario conectar varios dispositivos a una placa Arduino por sendos canales serie, mejor utilizar siempre la comunicación por hardware con el dispositivo que genere un mayor tráfico de datos.

USO DE MOTORES

Conceptos básicos sobre motores

La mayoría de los motores funcionan gracias al principio de inducción. Esto quiere decir que cuando un cable conduce corriente, se genera automáticamente un campo magnético alrededor de él. Si colocamos una bobina (es decir, un cable enrollado) por la que pasa corriente entre dos polos imantados norte y sur, esta bobina será atraída por un polo y repelida por el otro (o viceversa), debido al campo

magnético que ella misma genera. Cuanta más intensidad de corriente atraviese la bobina, más grande será el campo magnético generado y por tanto más grande será la atracción o repulsión. Si colocamos la bobina además sobre un eje que puede girar, la rotación de la bobina (y por tanto, del campo magnético generado por ella) hará que vaya siendo atraída y repulsada alternativamente por cada polo exterior, generando así un "rebote" de la bobina entre los polos que dará lugar a un movimiento circular ininterrumpido del eje (mientras circule corriente por la bobina).

Existen varios datos a tener en cuenta cuando se utilizan motores en nuestros circuitos: uno de ellos es el voltaje al que pueden funcionar eficientemente, señalado por el fabricante. Con ese voltaje el motor podrá girar a su máxima velocidad, a más voltaje corre el riesgo de quemarse. A menor voltaje, el motor girará a menor velocidad.

Otro dato importante es el consumo de corriente que tienen. Este depende sobre todo de la carga que están arrastrando: a más carga, más corriente necesitan. Cada motor tiene una "corriente de paro", que es la corriente que consume cuando su giro se para por una fuerza opuesta. La corriente de paro es mucho más grande que la corriente de giro, que es la corriente consumida cuando no hay carga. Puede haber motores que consuman durante un breve tiempo casi su corriente de paro al arrancar, debido a la inercia de pasar de estar parados a moverse. La fuente de alimentación debería poder ofrecer la corriente de paro y algo más para evitar problemas. Si no es el caso, entonces se debe utilizar transistores (o más radicalmente, relés) para amplificar la corriente aportada por la fuente y así alimentar el motor convenientemente.

Otro dato es la resistencia eléctrica que ofrece el motor (como cualquier otro componente eléctrico), medida en ohmios. Otro dato más es la velocidad de giro del motor, normalmente medido en "rpm" (revoluciones por minuto).

Otro dato es el torque (también llamado par) del motor. El torque es una medida de la fuerza de empuje (de "tracción") que tiene el motor. Concretamente, representa la fuerza que tiene que realizar una rueda colocada sobre el eje del motor para poder arrastrar una carga situada a una cierta distancia de ese eje y que ejerza una fuerza opuesta al giro (normalmente, su peso). Si, por ejemplo, tenemos una carga que ejerce un peso de 1N situada a 1m del eje de giro, el torque necesario para moverla es de $1\text{N}\cdot\text{m}$. Si esa misma carga la tenemos a 0,5m, el torque será de $0,5\text{N}\cdot\text{m}$ (por tanto, el motor ha de realizar "menos esfuerzo" para moverla). Fijarse que ese mismo torque se obtendría también de situar a 1m una carga con la mitad del peso (0,5N). El torque que viene en los datasheets es el "torque de paro", que es el torque

EL MUNDO GENUINO-ARDUINO

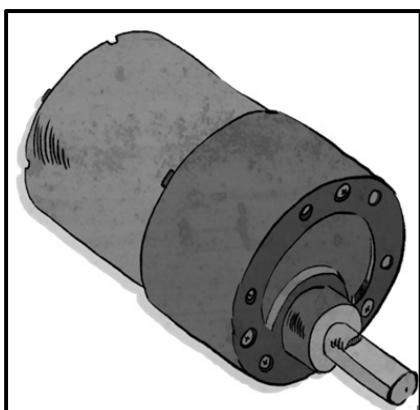
máximo que puede realizar el motor antes de no poder seguir girando el eje debido a la oposición de la carga: cuanto más torque tenga el motor, más cargas pesadas podrá rotar. Desgraciadamente, los fabricantes de motores no han estandarizado las unidades de medida del torque de paro, con lo que a veces nos podemos encontrar con datasheets que muestran esta magnitud en Kg·cm (kilogramo por centímetro), en vez de usar newtons como sería lo físicamente correcto.

Por otro lado, comentar que todos los circuitos inductivos (como los motores), además de inducir (es decir, "crear") un campo magnético gracias a una corriente eléctrica, también funcionan en el sentido contrario. Es decir, si existe un campo magnético variable (ha de ser así) donde hay una bobina, se induce automáticamente una corriente eléctrica a través de ella. Por lo tanto, si tenemos un motor girando y lo apagamos, el campo magnético que aún existe durante un breve tiempo inducido por la bobina rotatoria (y que es variable porque la bobina aún no ha parado de girar) hace que se genere una corriente en sentido contrario al de la corriente utilizada para mover el motor. Esta corriente puede ser muy intensa y dañar la electrónica, por lo que casi siempre veremos un diodo conectado en paralelo al motor para (al estar polarizado en inversa) parar esta corriente. Este diodo se suele llamar diodo "fly-back".

Tipos de motores

Cuando se quieren mover cosas con un microcontrolador, hay básicamente tres tipos de motores útiles: motores DC, servomotores y motores paso a paso (steppers).

Los motores DC



Los motores DC (del inglés "Direct Current", corriente continua) son los más simples. Tienen dos terminales; cuando uno se conecta a una fuente de alimentación continua y el otro se conecta a tierra el motor gira en una dirección. Si se intercambia la conexión de los terminales (el que estaba conectado a tierra pasa a estar conectado a la fuente, y viceversa), el motor girará en la dirección contraria. Cuanta más intensidad de corriente atravesé el motor (es decir, cuanto más voltaje se le aplique, si suponemos su resistencia constante), girará a

más velocidad de una forma casi linealmente proporcional.

CAPÍTULO 5: LIBRERÍAS ARDUINO

En general, los motores DC realizan un consumo eléctrico bastante elevado para conseguir la velocidad de giro adecuada. Esto quiere decir que muchas veces el pin de "5V" de nuestra placa Arduino no será suficiente, y el motor deberá ser alimentado a partir de una fuente externa, o bien mediante un amplificador de corriente (como un transistor).

Usualmente, los motores DC son capaces de girar hasta varios millares de rpms. No obstante, no tienen un torque demasiado elevado. Si se desea aumentar este, se puede conectar al motor un conjunto de engranajes (lo que se llama un "reductor" o "caja reductora"); el precio a pagar es la reducción de la velocidad máxima de giro. Los motores que incorporan este sistema son llamados motores "gearhead" o "garmotors".

Sparkfun distribuye varios motores "gearhead" con diferentes características de dimensiones, peso, velocidad, torque y consumo. Por ejemplo, tenemos los productos nº **8910**, **8911**, **8912** y **8913**. Si queremos motores DC sin reductor, en Sparkfun distribuyen el producto nº 9608 y en Adafruit el nº **711**. Otro tipo de motores DC son los llamados "motores lineales", los cuales son capaces de generar movimiento de tracción sobre un riel, pero estos no los veremos.

Un gran inconveniente de los motores DC es que, aunque podemos controlar fácilmente la velocidad del motor, el sentido del movimiento siempre es el mismo. Ya hemos dicho que cambiar el sentido del movimiento del motor DC (ya sea rotativo o lineal) implica alternar la ubicación de los terminales de la fuente de alimentación, cosa que es bastante poco práctica en la realidad. Para conseguir esto de una forma sencilla, lo que deberemos hacer es conectar el motor a un conjunto de transistores (o relés) –en todo caso, cuatro como mínimo– dispuestos de una determinada forma comúnmente llamada "puente H". Este nombre proviene de la figura con la que las conexiones de este circuito de transistores aparecen dibujadas en los esquemas eléctricos.

Existen muchas implementaciones concretas de "puentes H" ampliamente tratadas en los libros de electrónica, pero está fuera del alcance de este libro discutir su diseño. Lo que sí haremos es utilizar "puentes H" ya integrados dentro de chips. Estos chips ahoran el trabajo de montar el diseño "a mano" y simplemente ofrecen una serie de patillas donde conectar los terminales del motor convenientemente según lo que marque su datasheet concreto.

Un ejemplo muy utilizado de estos "puentes H" integrados es el chip L298 de STMicroelectronics, que es precisamente el chip que incorpora el Arduino Motor

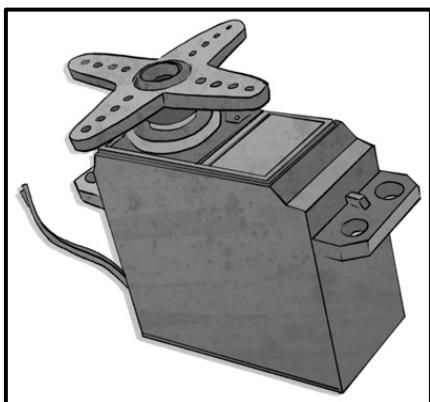
EL MUNDO GENUINO-ARDUINO

Shield oficial. En realidad, este chip incluye dos "puentes H", por lo que con él se puede controlar la velocidad y sentido de giro de dos motores DC (sin sobrepasar los 2A de corriente por cada puente, que es la máxima admitida) o incluso conectar ambos "puentes H" en paralelo para obtener así un solo "puente H" con el doble de capacidad de corriente.

El chip L298 se puede adquirir en la tienda oficial de Arduino o en la mayoría de distribuidores listados en el apéndice A (por ejemplo, en Sparkfun es el producto nº 9479). En Sparkfun también se puede adquirir con nº 9670 una placa breakout que incluye este chip, para una mayor comodidad de uso. Otro chip similar es el L293, el cual es más sencillo de conectar a una breadboard pero admite menos corriente de salida (0,6A) y disipa peor el calor; además su disposición de patillas no es compatible con la del L298. Una alternativa al chip L293 con su misma disposición de pines pero permitiendo el doble de corriente de salida (1,2A) es el chip SN754410.

El estudio práctico de los motores DC lo relegaremos al siguiente capítulo, donde estudiaremos el funcionamiento de las entradas y salidas digitales en una placa Arduino.

Los servomotores



Los servomotores –también llamados "servos"– son motores "gearhead" (por tanto, motores DC con engranajes que limitan la velocidad pero aumentan el torque) que incorporan además un potenciómetro y cierta circuitería de control para poder establecer la posición del eje del motor de forma precisa. Es decir, su eje no gira libremente (como lo hace el de los motores DC), sino que rota un determinado ángulo, indicado a través de una señal de control. Lo que hace especial a un servo es, por tanto, que podemos ordenarle

que gire una cantidad de grados concreta, cantidad que dependerá de la señal de control enviada en un momento dado por (por ejemplo) un microcontrolador programado por nosotros. Los servos son muy comunes en juguetes y otros dispositivos mecánicos pequeños (como por ejemplo el control de la dirección de un coche teledirigido), pero también sirven para gestionar el movimiento de timones, pequeños ascensores, palancas, etc.

Los servomotores disponen normalmente de tres cables: uno para recibir la alimentación eléctrica (normalmente de color rojo), otro para conectarse a tierra (normalmente de color negro o marrón, según el fabricante) y otro (el cable de control, normalmente de color blanco, amarillo o naranja) que sirve para transmitir al servo, de parte del microcontrolador, los pulsos eléctricos –de una frecuencia fija de 50Hz en la gran mayoría de servomotores– que ordenarán el giro concreto de su eje. El cable de alimentación ha de conectarse a una fuente que pueda proporcionar 5V y al menos 1A. El cable de tierra ha de conectarse lógicamente a la tierra común del circuito. El cable de control debe conectarse a algún pin digital de la placa Arduino, por el cual se enviarán los pulsos que controlarán el desplazamiento angular del eje. A diferencia de los demás motores DC, para cambiar el sentido de giro del eje de los servos no es necesario invertir la polaridad de su alimentación, por lo que no es necesario incluir ningún "puente H".

Los conectores del servo son ligeramente diferentes según el fabricante, pero todos suelen ser compatibles para el uso en breadboards y similares.

Los servomotores que utilizaremos en los proyectos de este libro son los más pequeños (los de tipo llamado "hobby"). Su voltaje de trabajo es de entre 5V y 7V, así que con la propia placa Arduino los podemos alimentar. No obstante, el consumo eléctrico de un servo es proporcional a la carga mecánica que soporta su eje (es decir, un servo consume más cuanta más "fuerza" –técticamente, torque– necesite generar para contrarrestar la masa de los objetos colocados sobre su eje y que se oponen a su giro). Esto significa que, en la práctica, dependiendo de la carga que se le coloque al servo, será necesario utilizar una fuente externa de 5V adicional independiente para proporcionarle una alimentación separada de la ofrecida a través de la placa Arduino (pero con la tierra común siempre). La fuente adicional también será necesaria cuando se empleen más de dos servos en nuestros circuitos, tengan la carga que tengan.

La magnitud del desplazamiento angular del eje de un servomotor está determinada por la duración de los pulsos de la señal de control. Concretamente, si el valor ALTO (5V) del pulso se mantiene durante 1,5 milisegundos, el eje del servo se ubicará en la posición central de su recorrido. Como los servos estándar permiten mover su eje en ángulos dentro de un rango entre 0 y 180 grados, esta posición central suele corresponder a 90 grados respecto al origen. Es decir: si al servomotor se le envía una señal con pulso de 1,5ms, el eje girará hasta estar situado en un ángulo de 90 grados respecto al origen (por tanto, a mitad de su recorrido total). Mientras la señal de control recibida por el servomotor sea siempre la misma, este mantendrá la posición angular de su eje: si la duración del pulso de la señal varía, entonces el servomotor girará hasta la nueva posición.

EL MUNDO GENUINO-ARDUINO

Si el ancho del pulso está entre 1,5 y 2 milisegundos, el eje del servo se moverá hasta una posición angular proporcional entre 90 y 180 grados del origen. Por ejemplo: si lo quisiéramos situar a 150 grados, la longitud del pulso debería ser de 1,83ms; si quisiéramos situarlo a 180 grados (ángulo máximo), la longitud del pulso debería ser de 2ms (duración máxima).

Si el ancho del pulso está entre 1 y 1,5 milisegundos, el eje del servo se moverá a una posición angular proporcional entre 0 y 90 grados del origen. Por ejemplo: si lo quisiéramos situar a 30 grados, la longitud del pulso debería ser de 1,17ms; si quisiéramos situarlo a 0 grados (es decir, el propio origen), la longitud del pulso debería ser de 1ms (duración mínima).

De todas formas, existen modelos de servos que llegan hasta los 210 grados de rotación, por lo que, en todo caso, se recomienda consultar los datos ofrecidos por el fabricante para conocer las características de cada servo en particular.

Por otro lado, hemos de destacar también la existencia de los servomotores de rotación continua, los cuales son algo especiales. En realidad, se comportan más como motores "gearhead" que como servos propiamente dichos, porque no se les puede establecer su ángulo de giro, pero, en cambio, sí su velocidad de giro. Lo más interesante es que podemos cambiar su sentido del giro sin necesidad de ninguna circuitería extra (como serían los "puentes H" o similares).

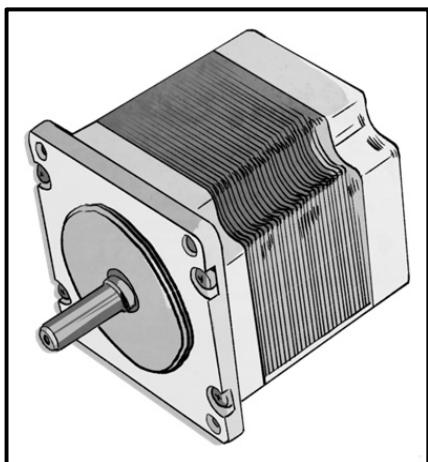
La señal de control de un servomotor de rotación continua está formada por pulsos cuadrados, generalmente a una frecuencia de 50Hz (por tanto, igual que la de los servomotores estándar). Cuando la duración del valor ALTO (5V) de un pulso se mantiene durante 1,5ms, un servomotor de rotación continua permanece parado. A medida que aumenta esa duración, su velocidad de giro aumenta en un determinado sentido, hasta llegar a la velocidad máxima cuando el valor ALTO del pulso llegue a durar un máximo de 1,7ms (generalmente). Si, por el contrario, la duración del valor ALTO del pulso es menor de 1,5ms, el giro se producirá en el sentido contrario, y su velocidad irá en aumento a medida que la longitud del pulso disminuya, hasta llegar a durar un mínimo de 1,3ms (generalmente). Por ejemplo, si un servomotor de este tipo recibe una señal con un pulso de 1,525ms, girará lentamente en un sentido, y si recibe una señal con un pulso de 1,575ms girará en el mismo sentido pero algo más deprisa; si recibe un pulso de 1,395ms, girará más deprisa aún, pero en el otro sentido.

Los servos se pueden adquirir fácilmente en cualquier distribuidor electrónico como los que están listados en el apéndice A. También se puede consultar

<http://www.servocity.com>, un sitio web especializado en servomotores y motores en general. Como ejemplo de productos podemos nombrar los ofrecidos por Adafruit: el nº 169 (microservo), el nº 155 (servo estándar), el nº 154 (servo de rotación continua) o el kit nº 171, ideal para empezar a trabajar con motores, ya que incluye el microservo y el servo estándar anteriores, además de un motor DC, un motor stepper y el Arduino Motor Shield. Sparkfun ofrece por su parte los productos nº 9065 (microservo), nº 9064 (servo estándar) y los nº 9347 y nº 10189 (servos de rotación continua).

Destaquemos por otro lado la iniciativa de <http://www.openservo.com>, proyecto que ofrece a la comunidad la posibilidad de construir un servomotor digital totalmente libre. Para ello pone a disposición tanto los diseños de PCB como el firmware necesario para poder fabricar un servomotor propio.

Los motores paso a paso



Los motores "paso a paso" (en inglés, "stepper motors") se diferencian del resto de motores vistos anteriormente en que no giran continuamente, sino que lo hacen un número de "pasos" muy concretos. Un paso es el movimiento mínimo que puede hacer de una vez el motor, y su magnitud es configurable: puede consistir en una rotación tan grande como 90 grados, o bien un movimiento angular tan pequeño como 2 grados, según lo que interese en cada momento. Por ejemplo, para completar un giro de 360 grados, se necesitarían 4 pasos en el primer caso y 180 en el segundo. Como este diseño permite un control muy preciso de los movimientos del eje, estos motores son usados ampliamente en todo dispositivo donde la posición exacta del motor sea un requisito necesario, como por ejemplo impresoras, discos duros, etc.

control muy preciso de los movimientos del eje, estos motores son usados ampliamente en todo dispositivo donde la posición exacta del motor sea un requisito necesario, como por ejemplo impresoras, discos duros, etc.

Los motores paso a paso se mueven usualmente mucho más lentamente que los motores DC, ya que hay un límite máximo para la velocidad a la que se pueden ir dando los pasos, pero ofrecen mayor torque. De hecho, cuando están parados (pero siguen recibiendo alimentación eléctrica), los steppers ofrecen un torque muy elevado porque sus bobinas internas ejercen de freno, con lo que resulta muy difícil mover su eje. Existen dos tipos de motores paso a paso:

EL MUNDO GENUINO-ARDUINO

Unipolares: tienen cuatro bobinas internas (en realidad, no son cuatro bobinas: son solo dos, cada una de las cuales está dividida por una conexión central común por donde recibe la alimentación) rodeando el eje central, el cual está imantado. Del motor salen seis cables: cuatro de ellos corresponden a los extremos de las dos bobinas y los otros dos a sus respectivas conexiones centrales comunes. Puede haber modelos de motor que solo ofrezcan cinco cables: en ese caso, una de las conexiones centrales se conecta internamente a la otra y ambas se alimentan por tanto a partir de un único cable.

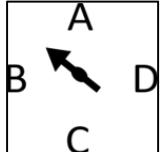
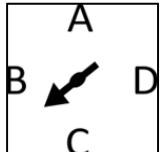
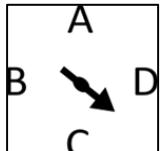
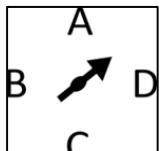
Para saber qué cable es cada cual, se debería de consultar el datasheet, aunque hay un método "manual" para averiguarlo: medir con un multímetro la resistencia existente entre los extremos de dos de los cables. La resistencia entre los cables extremos de cada bobina es el doble que la existente entre uno de esos cables y el de la conexión central. Y la resistencia entre cables pertenecientes a bobinas diferentes es infinita. Sabiendo esto, la averiguación es sencilla.

Bipolares: tienen dos bobinas internas independientes (que no están divididas por ninguna conexión central) rodeando el eje central, el cual está imantado. Del motor salen cuatro cables, que corresponden a los extremos de cada bobina. Los motores bipolares tienen aproximadamente un 30% más de torque que un motor unipolar del mismo volumen, pero su circuitería es algo más compleja.

El control del movimiento de los motores paso a paso se realiza aplicando cierto voltaje a sus bobinas (a través de los cuatro cables conectados en sus extremos respectivos), el cual ha de seguir un determinado patrón repetitivo. Según van recibiendo tensión o dejándola de recibir en cada paso de ese patrón, las bobinas crean los campos magnéticos adecuados para atraer o repeler los imanes del eje, causando así su rotación. Por tanto, mediante el patrón de corriente adecuado, podemos controlar el movimiento del motor al detalle.

Los patrones pueden ser muy variados. Un patrón típico podría ser, por ejemplo, el mostrado en la siguiente tabla. En cada paso de este patrón se activan dos bobinas, lo que provoca que el eje se quede situado entre ellas (las letras A, B, C y D representan las cuatro bobinas del motor). Después del paso 4 se volvería a empezar por el 1, y así sucesivamente. Es fácil deducir que invirtiendo el orden de un patrón, invertimos el sentido del movimiento.

CAPÍTULO 5: LIBRERÍAS ARDUINO

PASO	BOBINA A	BOBINA B	BOBINA C	BOBINA D	GIRO
1	ON	ON	OFF	OFF	
2	OFF	ON	ON	OFF	
3	OFF	OFF	ON	ON	
4	ON	OFF	OFF	ON	

Otro patrón bastante utilizado (aunque con menor torque de paso y de paro) es activar en cada paso una sola bobina; de esta forma, el eje se encará directamente a ella en vez de quedar en medio de dos. También se pueden combinar ambos para realizar un patrón de 8 pasos: A, A-B, B, B-C, C, C-D, D, D-A, etc.

La circuitería necesaria para poder enviar los patrones de señal correctos a las bobinas de un motor unipolar o bipolar (y así poderlos controlar) es relativamente compleja. En el caso de los motores bipolares, es necesario conectar a nuestro microcontrolador un "puente H" por cada bobina del motor, así que en realidad siempre necesitaremos dos "puentes H" iguales. En el caso de los motores unipolares, lo más habitual es utilizar un conjunto de 8 transistores de tipo Darlington, normalmente encapsulados dentro de un mismo chip, como por ejemplo el ULN2003: conectando de la forma adecuada las patillas del ULN2003 a nuestro microcontrolador, las cuatro bobinas podrán ser directamente gestionadas por este.

EL MUNDO GENUINO-ARDUINO

Como la mayoría de motores, los steppers requieren más electricidad de la que la placa Arduino les puede ofrecer, así que necesitaremos una fuente externa para alimentarlos. Existen muchos modelos de steppers, y cada uno de ellos trabaja a un determinado voltaje DC; para saber cuál es el voltaje de trabajo de nuestro motor deberíamos consultar el datasheet ofrecido por el fabricante. Valores típicos son 5V, 9V, 12V y 24V.

Adafruit distribuye los siguientes steppers: el producto **nº 858** (que funciona a 5V), el **nº 324** y el **nº 918** (que funcionan ambos a 12V). Sparkfun por su parte también distribuye diferentes steppers, como los **nº 10551, 9238, 10846, 10847 o 10848**, entre otros. En las especificaciones técnicas de cada motor se indica el número de pasos (N_p) que puede realizar como máximo en una vuelta, pero dependiendo del motor, hay que tener en cuenta que para obtener el número real de pasos posibles, el fabricante ha de indicar si N_p ha de ser multiplicado por el factor de reducción del tren de engranajes. Es decir, un motor de $N_p=48$ pasos que tenga 1/16 de factor de reducción en realidad podrá hacer $48 \cdot 16 = 768$ pasos por vuelta.

La librería Servo

Aunque podríamos controlar un servomotor "a pico y pala" generando directamente una señal digital de 50Hz con pulsos cuya duración variara entre 1ms y 2ms, es mucho más sencillo (y conveniente) utilizar la librería oficial "Servo". Lo primero que hay que hacer para poder controlar un servomotor utilizando esta librería es declarar un objeto de tipo Servo. Si suponemos que lo llamamos "miservo", esto se haría con la línea: Servo miservo; en la zona de declaraciones globales. A partir de aquí, la primera función imprescindible que debemos escribir (normalmente dentro de `setup()`) es:

miservo.attach(): vincula el objeto *miservo* con el pin digital de la placa Arduino donde está conectado físicamente el cable de control del servomotor. Su parámetro precisamente es el número de ese pin. Esta función no tiene valor de retorno.

A partir de aquí, ya podemos hacer uso del resto de funciones de la librería Servo para controlar los servomotores de nuestro circuito:

miservo.write(): controla el eje del servomotor. Su único parámetro es el ángulo (en grados) donde se quiere situar dicho eje. Como resultado de ejecutar esta instrucción en un servomotor estándar, se obtendrá un movimiento del eje hasta alcanzar ese ángulo especificado. En un servomotor

de rotación continua, en cambio, el valor de su parámetro representa la velocidad del eje: 0 equivale a la máxima velocidad en un sentido, 180 a la máxima velocidad en el sentido contrario y 90 equivale aproximadamente a inexistencia de movimiento. Esta función no tiene valor de retorno.

miservo.writeMicroseconds(): controla el eje del servomotor. Su parámetro –de tipo *int*–, en vez de ser el ángulo en grados como en *miservo.write()*, es la duración del pulso de la señal de control. Un valor de 1500 mantiene el eje en la posición central, un valor de 1000 lo mueve hasta su posición mínima y un valor de 2000 lo mueve hasta el otro extremo. Algunos fabricantes no siguen este estándar muy estrictamente, por lo que los servomotores a menudo responden a valores entre 700 y 2300, así que en la práctica deberemos ir probando de aumentar los límites hasta que el servomotor no aumente su movimiento. De todas formas, intentar hacer funcionar un servomotor más allá de sus límites (frecuentemente notificado por un ruido característico) es un estado de alta corriente y debería ser evitado. Los motores de rotación continua responden a esta función de la misma manera que con *miservo.write()*: variando su velocidad de rotación entre un mínimo y un máximo por cada sentido. Esta función no tiene valor de retorno.

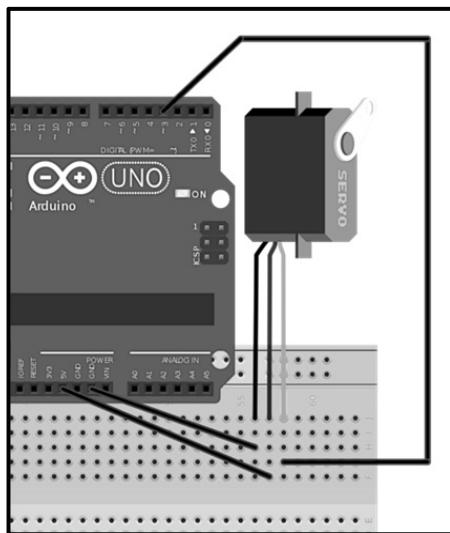
miservo.read(): devuelve el ángulo actual del servomotor (es decir, el valor pasado en la última ejecución de *miservo.write()*). No tiene ningún parámetro.

miservo.attached(): comprueba si el objeto *miservo* está vinculado a algún pin de la placa Arduino. Si es así, devuelve "true", si no, *false*. No tiene ningún parámetro.

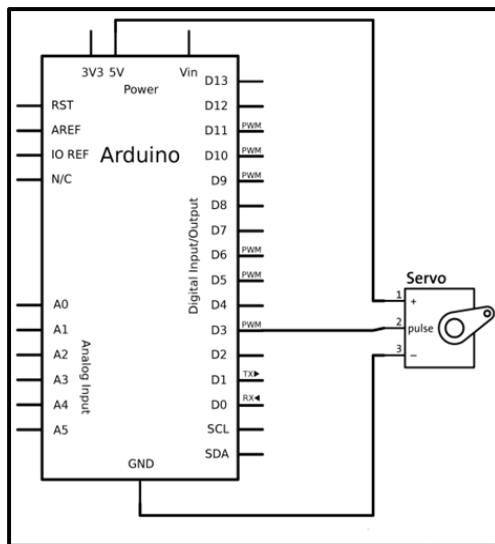
miservo.detach(): desvincula el objeto *miservo* del pin de la placa Arduino asociado. Solamente cuando todos los objetos Servo existentes en nuestro sketch sean desvinculados, los pines 9 y 10 de la placa podrán ser utilizados como salida PWM otra vez. No tiene ni parámetros ni valor de retorno.

Veamos unos ejemplos de uso. Conectemos un servo a nuestra placa Arduino tal como se muestra a continuación (nota: el pin digital donde se conecta el cable de control del servomotor NO tiene por qué ser de tipo PWM):

EL MUNDO GENUINO-ARDUINO



El diseño anterior tendría un esquema eléctrico tal como el siguiente:



El siguiente código mueve de forma continua el eje del servomotor desde un ángulo de 10 grados hasta 170 y seguidamente lo mueve en sentido contrario de 170 grados a 10, y así ininterrumpidamente.

Ejemplo 5.19

```
#include <Servo.h>
Servo miservo;
byte i = 0;
void setup() {
    miservo.attach(3);
}
void loop(){
//Va de 10 a 170 grados en pasos de un grado
for(i = 10; i < 170; i++) {
    miservo.write(i);
//Esperamos a que el servo alcance la nueva posición
delay(10);
}
//De 170 a 10, pero la mitad de rápido
for(i = 170; i>=10; i--) {
    miservo.write(i);
    delay(20);
}
}
```

Otro ejemplo puede ser (con el mismo circuito anterior) ordenar al servomotor que se sitúe en un ángulo determinado especificado a través del canal serie:

Ejemplo 5.20

```
#include <Servo.h>
Servo miservo;
void setup() {
    Serial.begin(9600);
    miservo.attach(3);
}
void loop(){
    long angulo;
    if (Serial.available() > 0) {
        angulo=Serial.parseInt();
        if (angulo > 10 && angulo < 170){
            miservo.write(angulo);
            delay(15);
        }
    }
}
```

EL MUNDO GENUINO-ARDUINO

Supongamos que tenemos dos servomotores situados frente a frente, con una rueda anclada a cada servomotor. Con el siguiente código podremos mover las dos ruedas en el mismo sentido (adelante o atrás) o bien en sentidos contrarios (provocando así giros hacia la derecha o hacia la izquierda).

Ejemplo 5.21

```
#include <Servo.h>
Servo servoIzq;
Servo servoDer;
void setup() {
    servoIzq.attach(10);
    servoDer.attach(9);
}
void loop() {
    adelante();    delay(2000);
    atras();       delay(2000);
    giroDerecha(); delay(2000);
    giroIzquierda(); delay(2000);
    parar();      delay(2000);
}
void adelante() { servoIzq.write(0); servoDer.write(180); }
void atras() { servoIzq.write(180); servoDer.write(0); }
void giroDerecha() { servoIzq.write(180); servoDer.write(180); }
void giroIzquierda() { servoIzq.write(0); servoDer.write(0); }
void parar() { servoIzq.write(90); servoDer.write(90); }
```

Respecto a las conexiones a realizar para poner en funcionamiento el ejemplo anterior en la vida real, hay que tener en cuenta que alimentar dos servomotores por el pin de 5V de la placa Arduino puede ser un poco justo por el consumo que estos realizan, así que lo recomendable es conectar el cable de alimentación de los servomotores al terminal positivo de una fuente externa (como por ejemplo una pila de 9V), y el cable de tierra de los servomotores al terminal negativo de dicha fuente. Al hacer esto, es muy importante no olvidarse de unir las dos tierras existentes en el circuito: la de la fuente externa (representada por su terminal negativo) y la de la placa Arduino (tierra que en general es diferente de la primera porque la placa se suele alimentar de una fuente diferente: el cable USB); si no compartimos las dos tierras para que sean una sola, el comportamiento del circuito puede ser muy errático. Por otro lado, lógicamente cada servomotor deberá conectarse además a un pin digital de la placa Arduino (en el sketch son los nº 9 y nº 10).

La librería Stepper

Los motores paso a paso, tal como se han comentado, requieren el uso de un conjunto de transistores Darlington (si son unipolares) o de dos "puentes H" (si son bipolares). En el primer caso, se pueden utilizar chips integrados como el ULN2003 o el ULN2004 (la diferencia entre ambos está en el valor de la resistencia de sus entradas) o también el ULN2803 o el ULN2804 (con una salida más que los anteriores), o similares. En el segundo caso, se pueden utilizar chips integrados como el L293, el L298 o el SN754410, los cuales admiten diferentes grados de corriente de salida.

No obstante, como el diseño de circuitos usando estos chips integrados es relativamente complejo y varía según el modelo elegido de chip (ya que cada uno tiene una disposición de pines diferentes), en este libro no implementaremos "a mano" ningún circuito de control de steppers, y optaremos por utilizar shields o módulos específicos que ya tengan incorporados toda esta circuitería necesaria. De esta manera, tan solo nos tendremos que preocupar de conectar convenientemente los cables del motor paso a paso (4 o 5/6 según si es bipolar o unipolar, respectivamente) a los zócalos adecuados del shield/módulo y de alimentarlo: recordemos que los motores paso a paso necesitan alimentación externa (con 9V o 12V normalmente ya es suficiente, pero dependerá del modelo) porque los 5V que ofrece la placa Arduino se quedan cortos.

Si el lector aún desea construir un circuito de control de motores paso a paso desde cero, comentaremos simplemente que para el caso de un motor paso a paso unipolar podemos consultar dos diagramas disponibles en la página oficial de Arduino (<http://arduino.cc/en/Reference/StepperUnipolarCircuit>). En ambos diagramas interviene el chip ULN2004, pero en uno se utilizan dos pines-hembra de la placa Arduino como salidas digitales (uno por cada bobina) y en el otro, más sencillo, se utilizan cuatro (uno por cada extremo de cada bobina). Para el caso de un motor paso a paso bipolar, podemos consultar otros dos diagramas en <http://arduino.cc/en/Reference/StepperBipolarCircuit>. En ambos diagramas aparece el chip L293D (o equivalente), pero en uno se utilizan dos pines de la placa Arduino y en otro se utilizan cuatro.

No obstante, la librería oficial Stepper nos ofrece la posibilidad de controlar un motor paso a paso de una forma más sencilla. De hecho, solo tiene tres funciones:

EL MUNDO GENUINO-ARDUINO

Stepper(): devuelve un objeto de tipo Stepper (lo llamaremos *mistepper*) que representa un determinado motor paso a paso conectado a la placa Arduino. Este objeto lo utilizaremos en nuestro sketch para controlar dicho motor. La función *Stepper()* tiene la particularidad de que ha de ser escrita al principio del código de nuestro sketch (en la zona de declaraciones de variables globales, fuera por tanto incluso de *setup()*). Tiene varios parámetros: el valor del primero –de tipo *int*– es el número de pasos en una vuelta que es capaz de dar el motor como máximo. Este dato nos lo tiene que ofrecer el fabricante. No obstante, a veces lo que nos ofrece es otro dato: el número de grados por paso; si es así, podemos obtener fácilmente el número de pasos totales si dividimos 360 entre los grados por paso (es decir, si un motor tiene por ejemplo 3,6 grados por paso, es de $360/3,6=100$ pasos). El segundo y tercer parámetro –de tipo *int* también– representan los dos pines digitales de la placa Arduino donde se conectarán el motor si el diseño del circuito es con solo dos pines; si el diseño requiriera cuatro pines, se debería utilizar un cuarto y quinto parámetro extra que representan esos otros dos pines más de la placa Arduino.

***mistepper.setSpeed()*:** establece la velocidad del motor en vueltas por minuto (rpm). Esta velocidad se especifica como parámetro, el cual, aunque es de tipo *long*, ha de ser positivo. Esta función no hace que el motor gire: solamente especifica su velocidad en el momento que este empiece a girar (gracias a la ejecución de *mistepper.step()*). No tiene valor de retorno.

***mistepper.step()*:** gira el motor un número determinado de pasos, especificado como parámetro –de tipo *int*–. Si este número es positivo, girará en un sentido, y si es negativo, en el otro. La velocidad de giro viene determinada por la última ejecución de *mistepper.setSpeed()*. Atención: esta función es bloqueante. Esto quiere decir que el sketch se esperará hasta que el motor haya acabado de moverse para continuar ejecutándose. Por lo tanto, si por ejemplo a un motor de 100 pasos le hacemos girar 100 vueltas a 1 rpm, *mistepper.step()* tardará un minuto completo en acabar, bloqueando mientras tanto el resto del sketch. Para un mejor control, por tanto, se recomienda mantener una velocidad elevada y pocos pasos. Esta función no tiene valor de retorno.

Suponiendo que tenemos un motor paso a paso (unipolar o bipolar, da igual) conectado a nuestra placa Arduino (usando dos o cuatro cables), podemos ejecutar el siguiente sketch para observar su giro paso a paso lentamente en una dirección:

Ejemplo 5.22

```
#include <Stepper.h>
/*El valor del primer parámetro se tendrá que cambiar según el modelo de motor usado. Si se utiliza un
circuito con solo dos pines, el cuarto y quinto parámetro no se han de escribir.*/
Stepper miStepper(200, 8,9,10,11);
int contador = 0;
void setup() {}
void loop() {
    miStepper.step(1);
    delay(500);
}
```

El siguiente código, en cambio, hace girar el stepper una revolución entera en un sentido y otra en el otro:

Ejemplo 5.23

```
#include <Stepper.h>
//Suponemos un motor de 200 pasos y un circuito de 4 pines
Stepper miStepper(200, 8,9,10,11);
void setup() {
    miStepper.setSpeed(60); /*60 vueltas/minuto */
}
void loop() {
    //Realizo todos los pasos que hay en una vuelta entera
    miStepper.step(200);
    delay(500);
    //Ahora en sentido contrario
    miStepper.step(-200);
    delay(500);
}
```

También es posible cambiar la velocidad de giro del stepper (mediante *miStepper.setSpeed();*) especificando como parámetro un valor variable, obtenido por ejemplo de una lectura de algún sensor (lo veremos en el próximo capítulo).

En la wiki oficial de Arduino hay una librería alternativa para el control de motores paso a paso, algo más completa. Se llama "CustomStepper" y puede obtener de aquí: <http://arduino.cc/playground/Main/CustomStepper>. Otra librería alternativa, que también aporta más control que la oficial es la "AccelStepper" (<http://www.open.com.au/mikem/arduino/AccelStepper>). Ambas librerías permiten acelerar/desacelerar el movimiento del motor, la gestión independiente de múltiples motores, funciones no bloqueantes, mejor soporte a otros chips controladores, etc.

6 ENTRADAS Y SALIDAS

La utilidad más evidente de una placa Arduino es interaccionar con su entorno físico a través de sensores y actuadores. Para ello, disponemos de varias funciones que tratan señales de tipo digital (ya sean entradas o salidas) o de tipo analógico (ya sean entradas o salidas).

USO DE LAS ENTRADAS Y SALIDAS DIGITALES

Las funciones que nos ofrece el lenguaje Arduino para trabajar con entradas y salidas digitales son:

pinMode(): configura un pin digital (cuyo número se ha de especificar como primer parámetro) como entrada o como salida de corriente, según si el valor de su segundo parámetro es la constante predefinida INPUT o bien OUTPUT, respectivamente. Esta función es necesaria porque los pines digitales a priori pueden actuar como entrada o salida, pero en nuestro sketch hay que definir previamente si queremos que actúen de una forma o de otra. Es por ello que esta función se suele escribir dentro de *setup()*. No tiene valor de retorno.

Si el pin digital se quiere usar como entrada, es posible activar una resistencia "pull-up" de $20K\Omega$ que todo pin digital de las placas Arduino incorpora. Para ello, se ha de utilizar la constante predefinida INPUT_PULLUP en vez de INPUT, ya que la constante INPUT desactiva explícitamente estas resistencias "pull-ups" internas.

EL MUNDO GENUINO-ARDUINO

Recordemos que si un pin de entrada tiene su resistencia "pull-up" interna desactivada, en el momento que no esté conectado a nada puede recibir ruido eléctrico del entorno o de algún pin cercano y provocar así inconsistencias en los valores de entrada obtenidos (ya que estos cambiarán aleatoriamente en cualquier momento). Esto hace que por lo general sea recomendable activar la resistencia "pull-up".

NOTA1: Otra forma alternativa de activar la resistencia "pull-up", diferente de la descrita en el párrafo anterior, es utilizar la constante INPUT y además la función *digitalWrite()* de una forma muy concreta –ver NOTA2 en párrafos siguientes–. Por otro lado, otra forma diferente de conseguir el mismo resultado práctico sería utilizar, en vez de la resistencia "pull-up" interna, una resistencia "pull-down" extra externa (es decir, una resistencia que conectara ese pin a tierra); un valor de 10KΩ ya valdría.

Por otro lado, es importante recordar que si en un sketch empleamos el objeto *Serial* para establecer comunicación con la placa Arduino, entonces no podremos utilizar los pines 0 y 1 como entradas y salidas digitales, ya que estos pasarán a funcionar en exclusiva como pines RX y TX, respectivamente.

digitalWrite(): envía un valor ALTO (HIGH) o BAJO (LOW) a un pin digital; es decir, tan solo es capaz de enviar dos valores posibles. Por eso, de hecho, hablamos de salida "digital". El pin al que se le envía la señal se especifica como primer parámetro (escribiendo su número) y el valor concreto de esta señal se especifica como segundo parámetro (escribiendo la constante predefinida HIGH o bien la constante predefinida LOW, ambas de tipo *boolean*). Esta función no tiene valor de retorno.

Si el pin especificado en *digitalWrite()* ha sido previamente configurado –mediante *pinMode()*– como salida (que será lo más habitual), indicar la constante HIGH en *digitalWrite()* equivale a enviar a ese pin una señal de salida de 5V estables (o bien 3,3V en las placas que trabajen a ese voltaje) con una intensidad variable según el consumo que realice el dispositivo allí conectado (intensidad que, por otro lado, en una placa Arduino UNO no debería ser mayor de 40mA para no dañarla –ipero en otros modelos este límite es mucho menor!–); indicar la constante LOW, por su parte, equivale a enviar una señal de salida de 0V (y 0mA).

NOTA2: Si el pin está configurado como entrada, enviarle un valor HIGH equivale a activar la resistencia interna "pull-up" en ese momento (es decir, es idéntico a usar directamente la constante INPUT_PULLUP), y enviar un valor LOW equivale a desactivarla de nuevo.

En los casos en que los valores de voltaje y/o intensidad ofrecidos por un pin digital de salida en estado HIGH no sean lo suficientemente elevados para poder alimentar al dispositivo allí conectado (por ser este de alto consumo, como es el caso

de la mayoría de motores o de muchas matrices de LEDs, entre otros), este debería alimentarse con fuentes externas, evitando así daños en el pin (y en la placa entera) debido a un sobrecalentamiento. También puede ocurrir el caso contrario: que el valor HIGH deba ser reducido (mediante el uso de un divisor de tensión, de un convertidor de nivel o similar) para adecuarlo a un voltaje menor, correspondiente a la tensión de trabajo del componente allí conectado. En las páginas siguientes se irán viendo ejemplos de uno y de lo otro.

digitalRead(): devuelve el valor leído del pin digital (previamente configurado mediante *pinMode()* como entrada) cuyo número se haya especificado como único parámetro. Este valor de retorno es de tipo *int* y puede tener dos únicos valores (por eso, de hecho hablamos de entrada digital): la constante HIGH (1) o LOW (0).

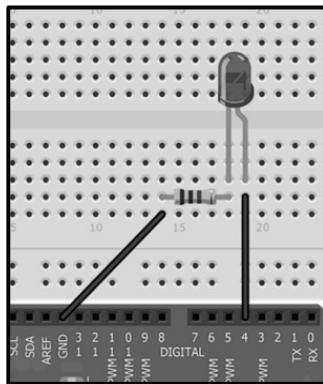
Si la entrada es de tipo INPUT, obtendremos el valor HIGH cuando la señal detectada en ella sea mayor de 3V y el valor LOW será reconocido cuando la señal recibida sea menor de 2V. Si la entrada es de tipo INPUT_PULLUP, al tener la entrada conectada una resistencia "pull-up", las lecturas son al revés: el valor HIGH indica que no se recibe señal de entrada y el valor LOW que sí. Por otro lado, hay que tener en cuenta la intensidad máxima que pueden absorber; en el caso de la placa Arduino UNO es de 40mA, pero en otros modelos es mucho menor.

Además de las anteriores, otra función con aplicaciones interesantes es:

pulseIn(): pausa la ejecución del sketch y se espera a recibir en el pin de entrada especificado como primer parámetro la próxima señal de tipo HIGH o LOW (según lo que se haya indicado como segundo parámetro) que llegue. Una vez recibida esa señal, empieza a contar los microsegundos que esta dura hasta cambiar su estado otra vez, devolviendo finalmente un valor –de tipo *long*– correspondiente a la duración en microsegundos de ese pulso de señal. De forma opcional, se puede especificar un tercer parámetro –de tipo *unsigned long*– que representa el tiempo máximo de espera en microsegundos: si la señal esperada no se produce una vez superado este tiempo, la función devolverá 0 y continuará la ejecución del sketch. Si este tiempo de espera no se especifica, el valor por defecto es de un segundo. En la documentación oficial recomiendan usar esta función para rangos de valores de retorno de entre 10 microsegundos y 3 minutos, ya que para pulsos más largos la precisión puede tener errores.

Ejemplos con salidas digitales

Veamos un código muy sencillo para ilustrar el uso de las funciones *pinMode()* y *digitalWrite()*. La idea es encender y apagar un LED de forma periódica, simplemente. El circuito montado en la breadboard sería parecido a este:



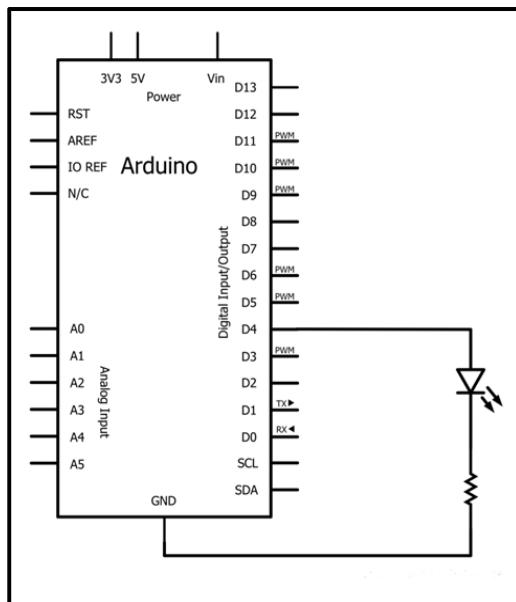
En él se puede observar que el terminal positivo del LED está conectado al pin-hembra digital número 4 de la placa Arduino y su terminal negativo está conectado a una resistencia que hace de divisor de tensión (un valor de 220 ohmios ya estaría bien), la cual está conectada a su vez a tierra.

Evidentemente, para que este circuito funcione, la placa Arduino ha de recibir alimentación eléctrica de alguna manera (en la figura anterior no se muestra la fuente porque para el caso es irrelevante). Una vez la placa esté encendida (y por tanto, nuestro sketch –mostrado en la página siguiente– esté ejecutándose), cuando así lo decida nuestro programa la placa enviará corriente al LED a través de su pin 4 configurado como salida (y por tanto, lo encenderá), y cuando "toque" dejará de enviarle corriente (y por tanto, lo apagará).

Notar que no se ha tenido que hacer uso de los pines-hembra "5V" o "Vin" de la placa Arduino; esto es un hecho que confunde mucho a los principiantes. Los pines-hembra de alimentación solamente son necesarios cuando queremos aportar electricidad de forma permanente y estable a algún componente de nuestro circuito (como un LED, por ejemplo) independientemente del estado de las salidas digitales. Tal como veremos en el código de ejemplo, para encender el LED enviamos un voltaje de 5V (el valor HIGH) a la salida digital en la que está conectado porque queremos tenerlo encendido solo cuando se envía precisamente ese valor HIGH, no todo el tiempo.

En cualquier caso, lo que sí que ha de tener todo circuito es una toma de tierra. Por tanto, en todos nuestros proyectos con Arduino siempre utilizaremos algún pin-hembra de los que están marcados como "GND".

El esquema correspondiente al circuito anterior será este:



Y he aquí finalmente el código:

Ejemplo 6.1

```
void setup(){
    //Inicializamos el pin digital 4 como salida. Es donde irá conectado nuestro LED.
    pinMode(4,OUTPUT);
}
void loop(){
    digitalWrite(4,HIGH); //Decimos que se encienda el LED
    delay(1000);          //Esperamos un segundo
    digitalWrite(4,LOW); //Decimos que se apague el LED
    delay(1000);          //Esperamos un segundo
}.
```

El sketch anterior es bastante autoexplicativo: tras establecer el pin digital número 4 como salida, enviamos a través de ella –gracias a la función *digitalWrite()*– una señal de valor HIGH (5V). Si no se hiciera nada en nuestro código para evitarlo,

EL MUNDO GENUINO-ARDUINO

esta señal, una vez activada, continuaría enviándose sin interrupción. Pero como lo que queremos es hacer parpadear el LED, justo después de enviar la señal de valor HIGH, enviamos una señal de valor LOW (0V) para así interrumpir el encendido. Una vez llegados al final de la función *loop()*, volvemos para arriba y continuamos el proceso: encendido-apagado-->encendido-apagado... y así sucesivamente.

¿Por qué intercalamos las funciones *delay()* en medio de las dos *digitalWrite()*? Porque si no estuvieran, el envío de la señal HIGH y el de la señal LOW se harían sumamente seguidos (ya que la velocidad con la que el microcontrolador pasa de ejecutar una línea del sketch a la siguiente es tremadamente alta; de hecho, eventos escritos en líneas contiguas dentro de la función *loop()* los podemos considerar prácticamente simultáneos) y nuestro ojo no podría distinguir ningún parpadeo. La función *delay()* sirve para pausar ("congelar") el sketch un determinado tiempo y así mantener la señal previamente enviada por el *digitalWrite()* justo anterior de forma que se pueda apreciar. Es evidente, pues, que escribir un tiempo menor en *delay()* hará que el parpadeo sea más rápido, y un tiempo mayor lo contrario. De hecho, si reducimos el parámetro de *delay()* hasta aproximadamente 10 milisegundos, veremos que efectivamente el LED deja de parpadear; esto es porque el parpadeo es tan rápido que nuestro ojo ya es incapaz de observarlo. Como dato curioso, si con 10ms de *delay()* además movemos nuestro montaje de un lado a otro dentro de una habitación oscura, veremos que el LED deja un camino de luz.

Teniendo lo anterior en cuenta, podríamos jugar un poco con el código. Por ejemplo, si cambiáramos el tiempo de espera de uno de los *delay()* podríamos mantener encendido o apagado el LED más tiempo o menos. Sabiendo esto (y añadiendo algún *digitalWrite()* y *delay()* más), podríamos realizar un ejercicio interesante: mostrar una secuencia de encendidos/apagados correspondientes a diferentes códigos Morse (tal como el S.O.S., compuesto por tres puntos, tres rayas y tres puntos), donde la raya equivaldría a un "encendido largo", el punto a un "encendido corto" y la pausa entre ellos a un apagado de duración única. Se deja como ejercicio.

En cualquier caso, para probar tanto el código 6.1 como el propuesto en el párrafo anterior (códigos muy comunes cuando se efectúa el primer contacto con el mundo Arduino, ya que el comportamiento de un LED es sencillo y fácilmente observable) es interesante saber que no es necesario utilizar ni siquiera ningún LED externo: solo con alimentar la placa ya será suficiente para comprobar si nuestro código funciona o no. Esto es así porque las placas Arduino incorporan de fábrica un LED interno pensado precisamente para realizar estas pruebas sin necesidad de requerir más componentes. En la placa Arduino UNO, ese LED está conectado

internamente siempre con la salida digital número 13, así que para poder utilizarlo deberíamos modificar la línea `pinMode(4, OUTPUT)`; de los ejemplos anteriores por la siguiente: `pinMode(13, OUTPUT)`. O mejor, como el número de salida puede ser diferente según el modelo de placa empleado (es decir, como en otras placas Arduino diferentes de la UNO el LED puede no estar conectado a la salida nº 13 sino a otra), para hacer más fácil el uso de este LED interno, como primer parámetro de `pinMode()` se puede indicar, en vez de un número explícito, la constante `LED_BUILTIN`, así: `pinMode(LED_BUILTIN, OUTPUT)`. Esta constante siempre vale, sea cual sea la placa utilizada, el número concreto del pin de salida conectado al LED interno; de esta manera, no nos tendremos que preocupar de acordarnos si ese pin era el nº 13 u otro.

Las salidas digitales no solo pueden comportarse de forma preprogramada (es decir, siguiendo un patrón fijo previamente definido, tal como se ve en el sketch 6.1), sino que también pueden ser controladas interactivamente, ya sea a través del canal serie, o bien mediante la pulsación de un botón o debido a la lectura de un determinado valor de un sensor, etc., etc. No será hasta el próximo apartado donde veremos varios ejemplos de interacción con LEDs mediante pulsadores (y no será hasta el capítulo 7 donde estudiaremos varios tipos de sensores), pero lo que sí podemos hacer ya es controlar la iluminación de un LED a través del canal serie, tal como, de hecho, hace el siguiente código: según la letra que reciba la placa Arduino, esta reacciona de una manera u otra (encendiendo el LED, apagándolo, haciéndolo parpadear, etc.).

Ejemplo 6.2

```
void setup() {
    Serial.begin(9600);
    pinMode(LED_BUILTIN, OUTPUT);
}
void loop() {
    //Espero la llegada de datos del canal serie. Si no llegan, no hago nada
    if(Serial.available() > 0) {
        char letra = Serial.read();
        //Hago algo diferente según el carácter recibido.
        switch (letra) {
            case 'e':
                digitalWrite(LED_BUILTIN, HIGH);
                break;
            case 'a':
                digitalWrite(LED_BUILTIN, LOW);
                break;
            case 'p':
```

EL MUNDO GENUINO-ARDUINO

```
//Para alargar el parpadeo, cambiar 10 por un nº mayor
for(byte i=1; i<=10; i++) {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(100);
    digitalWrite(LED_BUILTIN, LOW);
    delay(100);
}
break;
default:
    Serial.print("Órdenes posibles: 'e' (encender), 'a' (apagar), 'p' (parpadear)");
}
```

Evitando el uso de la función *delay()* –y de *delayMicroseconds()*–

El circuito del siguiente ejemplo es exactamente igual al utilizado en el ejemplo 6.1, y su comportamiento también (es decir, se consigue el parpadeo periódico de un LED). La novedad está en el código de nuestro sketch, donde, a riesgo de complicarlo un poco, no se utiliza ninguna función *delay()* para generar el tiempo de espera entre los estados encendido y apagado del LED. ¿Por qué este cambio si la función *delay()* parece muy sencilla y conveniente? Porque en cada *delay()* –y también en *delayMicroseconds()*– todo sketch se pausa ("se congela") hasta que termina el tiempo especificado y solo entonces continúa su ejecución. Esto provoca, por ejemplo, que si quisiéramos hacer parpadear un LED mientras quisiéramos a la vez estar pendientes de otras acciones (como, por ejemplo, la detección de la pulsación de un botón, o de la llegada de datos desde un sensor conectado a algún pin de entrada, o de la recepción de comandos transmitidos por el canal serie, etc., etc.) no podríamos usar *delay()* porque nuestro programa al llegar a la línea de esta instrucción se pararía completamente y se estaría "perdiendo" lo ocurrido "ahí fuera". Si, en cambio, evitamos la función *delay()* entonces sí podremos ejecutar instrucciones de nuestro código (como encender o apagar un LED) y al mismo tiempo realizar otras tareas (como recibir un dato del exterior, por ejemplo). Además, por otro lado, el uso de *delay()* conlleva un gasto de energía innútil al estar la placa Arduino en marcha sin hacer nada, aspecto que también es importante tener en cuenta.

El truco para no usar *delay()* –o *delayMicroseconds()*–, tal como se puede ver en el código siguiente, es ir obteniendo en cada vuelta del *loop()* el tiempo actual y comprobar, tras cada una de esas "tomas", si ya ha pasado el tiempo suficiente desde que ocurrió el último cambio de estado del LED para poder realizar un nuevo cambio. Si no es así, no hay nada que hacer, pero si es así, se realiza efectivamente el cambio

de estado del LED y se establece ese instante como "marca de último cambio realizado" para empezar así otra vez la cuenta de tiempo de cara al siguiente cambio que ha de llegar. Evidentemente, esta idea se puede generalizar para gestionar la periodicidad de todo tipo de eventos.

Ejemplo 6.3

```

boolean estadoLed = LOW;           //Variable que sirve para cambiar el estado del LED
unsigned long tActual = 0 ;        //Variable que guarda el instante actual de ejecución del sketch
unsigned long tUCambio = 0 ;        //Variable que guarda el último instante en el que el LED cambió
unsigned long intervalo = 1000;    //Variable que indica el intervalo de tiempo hasta el siguiente cambio
void setup(){
    pinMode(LED_BUILTIN,OUTPUT);
    Serial.begin(9600);
}
void loop() {
    tActual=millis(); //Obtengo el instante actual en esta iteración del loop
    /*Compruebo si ha llegado el instante de cambiar el estado del LED. Esto se hace mirando si
     la diferencia entre el instante actual –tActual– y el instante en el que el LED cambió por
     última
     vez –tUCambio– es igual o mayor que el intervalo deseado para el parpadeo. */
    if (tActual - tUCambio >= intervalo){
        /*Si es así, en la siguiente línea guardo el tiempo actual como el último instante de cambio
         (para usarlo en futuros loop)... Una línea alternativa también podría ser tUCambio =
         tUCambio + intervalo; ...*/
        tUCambio = tActual;
        //...y realizo el cambio de estado
        if (estadoLed == LOW){
            estadoLed = HIGH;
        } else {
            estadoLed = LOW;
        }
        digitalWrite(LED_BUILTIN, estadoLed);
    }
    /*A continuación viene el código que se desea ejecutar todo el rato (lectura de sensores,
     control de actuadores...) a la vez que el parpadeo de los LEDs. En este caso, como ejemplo,
     escribimos por el canal serie un mensaje repetidamente de forma fluida (sin parones) */
    Serial.println("Mensaje mostrado sin interrupciones una y otra vez");
}

```

El código anterior puede ser escrito de una forma mucho más clara si las líneas relacionadas con la cuenta de tiempos para efectuar el cambio del estado del LED se incluyen dentro de una función propia, que llamaremos *finIntervalo()*. De esta manera, al separar estas líneas del cuerpo principal del sketch, podremos reutilizarlas para gestionar cualquier otro tipo de cambio periódico que añadamos a nuestro

EL MUNDO GENUINO-ARDUINO

proyecto (tal como veremos en el próximo apartado). Concretamente, haremos que *finIntervalo()* devuelva *true* cuando sea el momento preciso de realizar un cambio y *false* si ese momento –periódico– aún no ha llegado (es decir, usaremos su valor de retorno como marca para señalar cada instante de cambio). Si además indicamos el intervalo de tiempo deseado entre cambio y cambio no como una variable global (como hacíamos en el sketch de ejemplo anterior), sino como un parámetro de *finIntervalo()*, ganaremos aún más flexibilidad. Todo esto se ve en el siguiente sketch:
NOTA: Otro cambio realizado en el sketch siguiente respecto al anterior es el haber definido la variable *tUCambio* dentro de la función *finIntervalo()* (y debido a esto, haberla declarado como *static* para mantener su valor entre llamada y llamada). Aunque esta variable podría haber continuado siendo de tipo global, con este cambio se ve más claramente que está ligada en exclusiva a esa función.

Ejemplo 6.4

```
boolean estadoLed = LOW;
void setup(){
    pinMode(LED_BUILTIN,OUTPUT);
    Serial.begin(9600);
}
void loop() {
    /*Ejecuto finIntervalo() y observo su valor de retorno para comprobar si ya se ha alcanzado el
    tiempo de espera indicado como parámetro (500ms en este caso). A cada iteración del loop()
    se vuelve a repetir el proceso.*/
    if(finIntervalo(500) == true){ //Si sí se ha alcanzado ese tiempo...
        if (estadoLed == LOW){ estadoLed = HIGH; } //...entonces realizo el cambio
        else { estadoLed = LOW; }
        digitalWrite(LED_BUILTIN, estadoLed);
    }
    //Se haya realizado el cambio o no, igualmente a continuación ejecuto el resto del sketch
    Serial.println("Mensaje mostrado sin interrupciones una y otra vez");
}

boolean finIntervalo(unsigned long intervalo){
    static unsigned long tUCambio = 0 ;
    unsigned long tActual=millis(); //Obtengo el instante actual
    /*Al igual que hacíamos dentro del if existente dentro de la función loop() del ejemplo 6.3,
    aquí comprobamos si la diferencia entre el tiempo actual y el del último cambio del LED es
    igual o mayor que el íntervalo dado.*/
    if (tActual - tUCambio >= intervalo){ //Si es así...
        /*...asigno el tiempo actual como tiempo de último cambio (el cual se guardará,
        al ser static, hasta la próxima llamada de finIntervalo())...*/
        tUCambio = tActual;
        return true; //...y devuelvo el valor que activa el cambio.
    }
    //Si esa diferencia es menor, no ha llegado aún el momento de cambiar...
} else {
```

```

        return false;           //...por lo que devuelvo un valor "inocuo"
    }
}

```

Es muy importante tener en cuenta que hasta que no se termina de ejecutar todo el código interior de *finIntervalo()* –así como, en general, de cualquier otra función propia que hayamos definido– no se procede a ejecutar las líneas siguientes presentes en el *loop()* del sketch. Es decir, la placa Arduino sigue esperando a que *finIntervalo()* finalice para proseguir con las siguientes tareas. La clave para que no se produzca el bloqueo que ocurría con *delay()* está en observar el interior de *finIntervalo()*: básicamente se compone de una ejecución de *millis()* y de una sección *if/else* muy simple (tan solo contiene asignaciones de variables); esto hace que sea muy rápida de procesar. Por tanto, el coste en tiempo de ejecutar *finIntervalo()* es ínfimo, lo que permite poder avanzar a las siguientes líneas del sketch casi inmediatamente.

De todas formas, aún podemos mejorar algo el código anterior: tal como en seguida veremos, es conveniente encapsular en otra función aparte (que llamaremos *cambioLed()*) también las líneas relacionadas con la acción repetitiva (en este caso, las relacionadas con el parpadeo del LED). De esta manera, tendremos confinado el comportamiento de ese elemento cambiante (el LED) respecto al resto del sketch, haciendo que su control sea más sencillo y menos propenso a errores.

NOTA: Una consecuencia de este cambio es que deberemos definir la variable *estadoLed* dentro de la función *cambioLed()* –y, además, debido a esto, deberá ser declarada como *static* para mantener su valor entre llamada y llamada–; aunque esta variable podría haber continuado siendo de tipo global, con este cambio se ve más claramente que está ligada en exclusiva a esa función.

Ejemplo 6.5

```

void setup(){
    pinMode(LED_BUILTIN,OUTPUT);
    Serial.begin(9600);
}
void loop() {
    if(finIntervalo(500) == true) {
        cambioLed();
    }
    Serial.println("Mensaje mostrado sin interrupciones una y otra vez");
}
boolean finIntervalo(unsigned long intervalo){
    static unsigned long tUCambio = 0 ;
    unsigned long tActual=millis();
    if (tActual - tUCambio >= intervalo){

```

EL MUNDO GENUINO-ARDUINO

```
tUCambio = tActual;  
    return true;  
} else {  
    return false;  
}  
}  
void cambioLed(){  
    static boolean estadoLed = LOW;  
    if (estadoLed == LOW){ estadoLed = HIGH; }  
    else { estadoLed = LOW; }  
    digitalWrite(LED_BUILTIN, estadoLed);  
}
```

Otro uso común de *delay()*/*delayMicroseconds()* es, además del de contar intervalos de tiempo para realizar tareas repetitivas, el de contarlos para retrasar una única ejecución de una determinada tarea aislada. En este caso, podemos volver a utilizar la misma técnica mostrada en los códigos anteriores para evitar el uso de *delay()*/*delayMicroseconds()* y, por tanto, para evitar el bloqueo de nuestra placa Arduino mientras dure la espera hasta, en este caso, la activación de esa tarea en cuestión. El siguiente código, por ejemplo, solamente ejecuta la función *encenderLed()* una vez —por tanto, el LED simplemente pasará de apagado a encendido y nada más— pero lo hace al cabo de cinco segundos de haberse iniciado la ejecución del sketch; no obstante, durante estos cinco segundos de espera el sketch no permanece en ningún momento bloqueado, por lo que mientras tanto puede ir enviando datos al canal serie sin interrupción (así como también, lógicamente, después de haberse encendido el LED, ya pasados esos cinco segundos). En este caso, el parámetro numérico de *finEspera()*—función en cierta medida asimilable, aunque diferente, a la función *finIntervalo()* del sketch anterior— ahora indica, en cambio, el tiempo de retraso en activar la acción:

Ejemplo 6.6

```
/*Cuando la variable 'espera' valga 'false', significará que la acción  
ya se ha disparado (y por tanto, que no se ha de volver a ejecutar más) */  
boolean espera = true;  
//tInicial' es necesaria para empezar a contar el tiempo desde el arranque del sketch  
unsigned long tInicial;  
void setup(){  
    pinMode(LED_BUILTIN,OUTPUT);  
    Serial.begin(9600);  
    tInicial = millis();  
}  
void loop() {  
    if(finEspera(5000) == true) { //Ejecuto la acción (no bloqueante) con un retraso de 5s
```

```

        encenderLed();
    }
    //La siguiente parte del sketch se ejecuta sin parar (antes y después de la acción retrasada)
    Serial.println("Mensaje mostrado sin interrupciones una y otra vez");
}

boolean finEspera(unsigned long retraso){
    unsigned long tActual=millis();
    /*Para ejecutar la acción retrasada se han de cumplir dos condiciones:
     que haya transcurrido el intervalo indicado y que, además, la variable 'espera' valga 'true' */
    if ((espera == true) && (tActual - tInicial >= retraso)) { return true; }
    else { return false; }
}
void encenderLed(){
    digitalWrite(LED_BUILTIN, HIGH); //Realizo la acción (encender el LED)
    //El valor 'false' de 'espera' hace que no se ejecute más el interior del if de llegoElMomento()
    espera = false;
}

```

Exactamente la misma técnica empleada en el código anterior puede ser aplicada en otro caso práctico bastante frecuente: para establecer un tiempo de espera máximo no bloqueante (un "timeout") ante la detección de algún determinado evento (como la pulsación de un botón o la llegada de un dato por el canal serie o por un pin de entrada), más allá del cual, si ese evento no ha sido recibido, el sketch dejará de estar pendiente de su detección. Se deja como ejercicio al lector.

Para finalizar, comentaremos que, si se prefiere, en vez de utilizar directamente las funciones *millis()* y *micros()* oficiales de Arduino, se puede optar por emplear alguna de las librerías de terceros que están especialmente diseñadas para realizar de una forma más intuitiva (gracias al uso de variables-contadores sencillos de manejar) la gestión no bloqueante de los diferentes tiempos de espera existentes en nuestro sketch, facilitando así la programación de ejecuciones de varias tareas en paralelo. Ejemplos de algunas de estas librerías (en ningún orden en particular) son:

- "Metro" (<https://github.com/thomasfredericks/Metro-Arduino-Wiring>)
- "Timer" (<https://github.com/JChristensen/Timer/tree/v2.1>)
- "ElapsedMillis" (<https://github.com/pfeerick/elapsedMillis>)
- "TaskScheduler" (<https://github.com/arkhipenko/TaskScheduler>)
- "ArduinoThread" (<https://github.com/ivanseidel/ArduinoThread>)
- "SCoop" (<https://github.com/fabriceo/SCoop>)
- "SoftTimer" (<https://github.com/prampec/arduino-softtimer>)
- "AsyncDelay" (<https://github.com/stevemarple/AsyncDelay>)
- "Chrono" (<https://github.com/thomasfredericks/Chrono>)

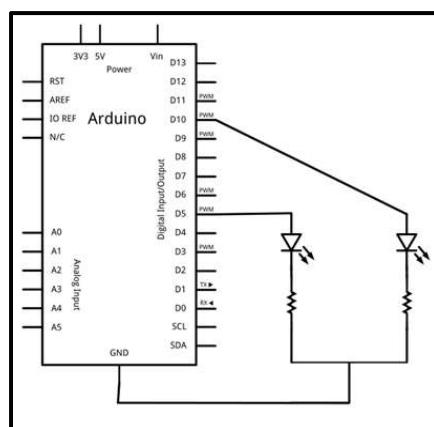
EL MUNDO GENUINO-ARDUINO

Por otro lado, recordemos que existe la librería oficial "Scheduler" (<https://www.arduino.cc/en/Reference/Scheduler>) pero solo puede ser usada en sketches ejecutados en la placa Arduino Due.

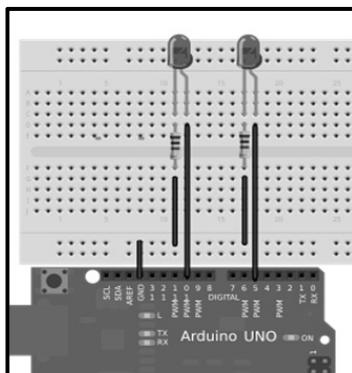
Múltiples salidas en paralelo

En los ejemplos anteriores nuestra placa Arduino solamente manipulaba una sola salida digital (concretamente conectada a un LED). En este apartado daremos un paso más y veremos diferentes circuitos donde nuestra placa Arduino se tendrá que encargar de enviar señales a varias salidas digitales (las cuales, también estarán conectadas a sendos LEDs, en aras de mantener los ejemplos lo más simples posibles).

Para empezar, estudiaremos un circuito con 2 LEDs conectados cada uno a un pin-hembra de la placa Arduino diferente (en nuestro ejemplo son el 5 y el 10, pero podrían ser otros cualesquiera). Su esquema técnico sería así...:



... y su aspecto sobre una placa de prototipado sería similar a este:



Una vez montado el circuito, podemos hacer que este se comporte de varias maneras dependiendo del sketch introducido en la placa Arduino. Por ejemplo, si queremos que ambos LEDs brillen y se apaguen a la vez, el sketch más simple para conseguir este efecto es el siguiente:

Ejemplo 6.7

```
void setup(){
    //Definimos como salidas los dos pines digitales donde están conectados sendos LEDs
    pinMode(5,OUTPUT);
    pinMode(10,OUTPUT);
}
void loop(){
    /*Enviamos (casi) simultáneamente dos señales HIGH a sendas salidas
     *y las mantenemos activas durante un segundo */
    digitalWrite(5,HIGH);
    digitalWrite(10,HIGH);
    delay(1000);
    /*Ahora enviamos (casi) simultáneamente dos señales LOW a sendas salidas
     *y las mantenemos activas durante un segundo */
    digitalWrite(5,LOW);
    digitalWrite(10,LOW);
    delay(1000);
}
```

También podríamos hacer que se iluminaran alternativamente:

Ejemplo 6.8

```
void setup(){
    pinMode(5,OUTPUT);
    pinMode(10,OUTPUT);
}
void loop(){
    digitalWrite(5,HIGH);
    digitalWrite(10,LOW);
    delay(1000);
    digitalWrite(5,LOW);
    digitalWrite(10,HIGH);
    delay(1000);
}
```

Tal como vimos en el apartado anterior, podríamos conseguir el mismo resultado que el logrado con los dos sketches anteriores sin necesidad de usar *delay()*

EL MUNDO GENUINO-ARDUINO

(y, por tanto, sin bloquear el funcionamiento de nuestra placa). Concretamente, para iluminar sincronizadamente los dos LEDs podemos ejecutar el siguiente código, muy parecido al 6.5 pero con algunos cambios, enumerados a continuación:

- Se han inicializado dos variables globales (*pinLedA* y *pinLedB*) cuyo valor representa el número de pin-hembra de la placa Arduino donde se conectarán sendos LEDs (en este caso, los pines-hembra 5 y 10). De esta manera, allí donde es necesario indicar ese número (en *pinMode()*, en todos los *digitalWrite()*, etc.) no tenemos que especificarlo directamente, sino que basta con utilizar su variable asociada. La ventaja de esto es que si en un futuro cambiáramos la conexión de alguno de los LEDs a otro pin-hembra, no tendríamos que ir repasando todo el sketch para cambiar su número concreto, sino que solo tendríamos que cambiar un único lugar del código: la inicialización de la variable correspondiente. Este hecho nos ahorrará mucho trabajo y nos evitará cometer posibles errores en las modificaciones (además de que clarifica nuestro código al dar un nombre significativo a cada número de pin-hembra); a lo largo del libro veremos diferentes ejemplos que hacen uso de esta técnica.
- Las funciones *finIntervalo()* y *cambioLed()* del sketch 6.5 (vinculadas en aquel sketch al único LED que había en el circuito de ejemplo) han sido renombradas a *finIntervalo1()* y *cambioLedA()*, respectivamente. Esto se ha hecho para señalar claramente que ahora hay un segundo LED, cuyo parpadeo estará controlado en exclusiva por otras dos funciones (nuevas) llamadas *finIntervalo2()* y *cambioLedB()*, análogas a las primeras. De hecho, el código interno de ambas funciones *finIntervaloX()* es idéntico, y el de las dos funciones *cambioLedX()* solo varía en la salida digital que controlan (es decir, al indicar el pin-hembra al que está conectado el LED respectivo). Uno podría pensar entonces en simplificar el código y usar una sola función *finIntervalo()* genérica para los dos LEDs (y otra función común llamada *cambioLed()* simplemente añadiéndole un parámetro indicando el número de salida digital a controlar) pero esto no funcionaría porque entonces solamente habría una sola variable *estadoLed* y otra *tUCambio* para los dos LEDs, algo que es inadmisible porque cada uno de ellos ha de guardar su propio estado y temporizador de forma completamente independiente. Observar, de todas formas, que las variables *tUCambio* y *estadoLed*, al ser *static* –y por tanto, locales a cada función–, pueden tener el mismo nombre en *finIntervalo1()* y *finIntervalo2()* y en *cambioLedA()* y *cambioLedB()* respectivamente, sin ningún tipo de problema al no haber solapamiento de nombres.

Ejemplo 6.9

```

byte pinLedA = 5;
byte pinLedB = 10;
void setup(){
    pinMode(pinLedA,OUTPUT);
    pinMode(pinLedB,OUTPUT);
    Serial.begin(9600);
}
void loop() {
    //Como el parámetro vale lo mismo en ambas funciones finIntervaloX()...
    //...ambos LEDs brillan sincronizadamente
    if(finIntervalo1(500) == true) { cambioLedA(); }
    if(finIntervalo2(500) == true) { cambioLedB(); }
    Serial.println("Mensaje mostrado sin interrupciones una y otra vez");
}
boolean finIntervalo1(unsigned long intervalo){
    static unsigned long tUCambio = 0 ;
    unsigned long tActual=millis();
    if (tActual - tUCambio >= intervalo){
        tUCambio = tActual;
        return true;
    } else { return false; }
}
boolean finIntervalo2(unsigned long intervalo){
    static unsigned long tUCambio = 0 ;
    unsigned long tActual=millis();
    if (tActual - tUCambio >= intervalo){
        tUCambio = tActual;
        return true;
    } else { return false; }
}
void cambioLedA(){
    static boolean estadoLed = LOW;
    if (estadoLed == LOW){ estadoLed = HIGH; }
    else { estadoLed = LOW; }
    digitalWrite(pinLedA, estadoLed);
}
void cambioLedB(){
    static boolean estadoLed = LOW;
    if (estadoLed == LOW){ estadoLed = HIGH; }
    else { estadoLed = LOW; }
    digitalWrite(pinLedB, estadoLed);
}

```

EL MUNDO GENUINO-ARDUINO

Si no nos gusta tener que definir dos funciones –una temporizadora y otra modificadora de estado– por cada salida a controlar (en este caso, dos funciones por LED hacen que tengamos que definir cuatro funciones diferentes pero casi idénticas) una opción sería fusionar ambas (*finIntervaloX()* y *cambioLedX()*) en una sola, de manera que cada salida estaría gestionada por lo que viene a llamarse técnicamente "máquina de estado finito". Es decir:

Ejemplo 6.9(BIS)

```
byte pinLedA = 5;
byte pinLedB = 10;
void setup(){
    pinMode(pinLedA,OUTPUT);
    pinMode(pinLedB,OUTPUT);
    Serial.begin(9600);
}
void loop() {
    tActual = millis();
    maqEstFinLed1(1000);      //Como el parámetro vale lo mismo en ambas funciones...
    maqEstFinLed2(1000);      //...ambos LEDs brillan sincronizadamente
    Serial.println("Mensaje mostrado sin interrupciones una y otra vez");
}
void maqEstFinLed1(unsigned long intervalo){
    static boolean estadoLed = LOW;
    static unsigned long tUCambio = 0 ;
    unsigned long tActual=millis();
    if (tActual - tUCambio >= intervalo){
        tUCambio = tActual;
        if (estadoLed == LOW){ estadoLed = HIGH; }
        else { estadoLed = LOW; }
        digitalWrite(pinLedA,estadoLed);
    }
}
void maqEstFinLed2(unsigned long intervalo){
    static boolean estadoLed = LOW;
    static unsigned long tUCambio = 0 ;
    unsigned long tActual=millis();
    if (tActual - tUCambio >= intervalo){
        tUCambio = tActual;
        if (estadoLed == LOW){ estadoLed = HIGH; }
        else { estadoLed = LOW; }
        digitalWrite(pinLedB, estadoLed);
    }
}
```

Para hacer que cada LED parpadee con un intervalo diferente del otro, bastará con cambiar, en el código anterior, el número indicado en la invocación de *maqEstFinLed1()* –o *maqEstFinLed2()*, según convenga– por el valor concreto deseado. Si lo que queremos, en cambio, es iluminar los dos LEDs alternativamente pero de forma sincronizada, el código a ejecutar será exactamente el mismo que el 6.9 o 6.9Bis excepto en que el valor inicial de una de las dos variables de estado *estadoLed* (tanto es) deberá ser HIGH.

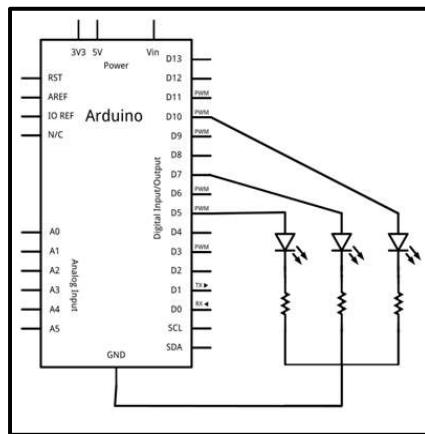
Algo más de trabajo, en cambio, lleva adaptar el código anterior para conseguir que cada LED tenga un tiempo de encendido diferente al de su apagado. Es decir, conseguir, por ejemplo, que el LED1 esté encendido durante 500ms y apagado durante 1s y el LED2 esté encendido durante 600ms y apagado durante 400ms. Se deja como ejercicio al lector, pero como pista mostraremos una sugerencia de cómo debería quedar el código de *maqEstFinLed1()* para conseguirlo:

```
void maqEstFinLed1(unsigned long intervaloON, unsigned long intervaloOFF){
    static boolean estadoLed = LOW;
    static unsigned long tUCambio = 0 ;
    unsigned long tActual=millis();
    //Si se ha superado el tiempo de encendido del LED, lo apago
    if ((estadoLed == HIGH) && (tActual - tUCambio >= intervaloON)){
        tUCambio = tActual;
        estadoLed = LOW;
        digitalWrite(pinLedA, estadoLed);
    //Si ya se ha superado el tiempo apagado del LED, lo enciendo
    } else if ((estadoLed == LOW) && (tActual - tUCambio >= intervaloOFF)){
        tUCambio = tActual;
        estadoLed = HIGH;
        digitalWrite(pinLedA, estadoLed);
    }
}
```

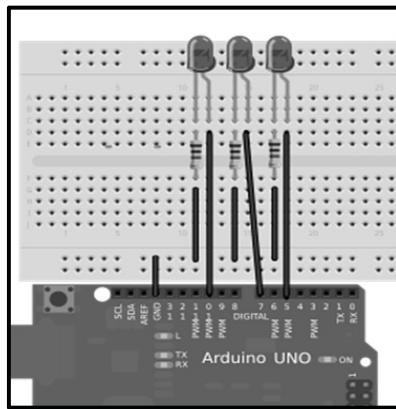
Finalmente comentaremos que, de forma similar a como ya hicimos anteriormente con el sketch 6.2, nada impide que podamos controlar interactivamente a través del canal serie cada LED de forma diferenciada tal como deseemos.

Pasemos ahora a estudiar otro circuito de ejemplo consistente en 3 LEDs conectados a sendos pines-hembra de la placa Arduino (en concreto, a los números 5, 7 y 10, pero podrían ser otros cualesquiera). Su esquema técnico sería así:

EL MUNDO GENUINO-ARDUINO



Y su aspecto sobre una placa de prototipado es este:



Hacer que se iluminen sincronizadamente los tres LEDs es una tarea trivial si partimos del código 6.9Bis: tan solo hay que añadir en *setup()* la línea *pinMode()* adecuada y en *loop()* invocar a una nueva función (llamada, por ejemplo, *maqEstFinLed3()*), cuyo código interno siga el mismo patrón que el de las ya conocidas *maqEstFinLed1()* y *maqEstFinLed2()*.

Hacer que cada LED parpadee a intervalos diferentes entre ellos también es trivial si seguimos empleando el código 6.9Bis como plantilla: tan solo hay que cambiar el valor numérico del parámetro de cada función (*maqEstFinLed1()*, *maqEstFinLed2()* y *maqEstFinLed3()*) indicando el intervalo de parpadeo deseado para cada LED en particular.

Otro ejercicio interesante —esta vez usando *delay()*— es encender secuencialmente los LEDs de tal forma que se simule el efecto "coche fantástico"; es decir, iluminar los LEDs por este orden de número de pin: 5, 7, 10, 7, 5, 7, 10... Para conseguir este efecto, el código del sketch ha de ser parecido al siguiente (notar que el truco está en utilizar un array para almacenar elementos cuyos valores son los números de pin donde están conectados los LEDs: de esta forma, utilizando los índices de esos elementos, se puede recorrer el array secuencialmente):

Ejemplo 6.10

```
//Este array tiene los nº de pines donde se conectan los LED
byte leds[]={5,7,10};    //leds[0]=5 ; leds[1]=7 ; leds[2]=10
byte i=0;                //Variable contador para los bucles for
int del=30;              //Variable que marca el tiempo en los delay()
void setup() {
    //El bucle recorre los tres elementos del array (0 -pin 5-, 1 -pin 7- y 2 -pin 10-)
    for (i=0;i<=2;i++) {
        pinMode(leds[i],OUTPUT);
    }
}
void loop() {
    //Primero se iluminan consecutivamente los tres LEDs en el orden: 5->7->10
    for (i=0;i<=2;i++) {
        digitalWrite (leds[i],HIGH);
        delay(del);
        digitalWrite (leds[i],LOW);
    }
    /*Seguidamente se iluminan consecutivamente solo los LEDs interiores (es decir, todos menos los dos de los extremos) haciendo la secuencia inversa para volver al principio y, en la siguiente iteración del loop(), poder volver a empezar de nuevo con el orden 5->7->10. En este caso particular, no habría hecho falta ningún bucle porque solo hay un LED entre ambos extremos (el que ocupa la posición 1 dentro del array) pero se deja como demostración para posibles futuras ampliaciones del proyecto. El contador i empieza valiendo 1 porque es la posición del penúltimo elemento del array (2-1=1) y acaba valiendo 1 porque es la posición del segundo elemento del array (0+1=1) */
    for (i=1;i>=1;i-) {
        digitalWrite (leds[i],HIGH);
        delay(del);
        digitalWrite (leds[i],LOW);
    }
}
```

Una variante interesante del código anterior es la que permite añadir una "estela" de luz que persigue a cada nuevo LED encendido. Concretamente, el sketch siguiente mantiene iluminados a la vez un LED y el siguiente del array a medida que va avanzando el ciclo, logrando así el efecto buscado:

EL MUNDO GENUINO-ARDUINO

Ejemplo 6.11

```
byte leds[]={5,7,10};  
byte i=0;  
int del=30;  
void setup() {  
    for (i=0;i<=2;i++) {  
        pinMode(leds[i],OUTPUT);  
    }  
}  
void loop() {  
    //Atención a los límites del bucle for: son diferentes a los del sketch anterior  
    for (i=0;i<2;i++) {  
        digitalWrite(leds[i],HIGH);  
        delay(del);  
        digitalWrite(leds[i+1],HIGH);  
        delay(del);  
        digitalWrite(leds[i],LOW);  
        delay(del*2);  
    }  
    //Atención a los límites del bucle for: son diferentes a los del sketch anterior  
    for (i=2;i>0;i--) {  
        digitalWrite(leds[i],HIGH);  
        delay(del);  
        digitalWrite(leds[i-1],HIGH);  
        delay(del);  
        digitalWrite(leds[i],LOW);  
        delay(del*2);  
    }  
}
```

Otro ejercicio, esta vez propuesto como ejercicio al lector, es escribir un sketch que simule el comportamiento de un semáforo (si los tres LEDs fueran, respectivamente, de color verde, naranja y rojo sería perfecto). Es decir, hacer que el LED "verde" se encienda durante un rato, que al cabo de un tiempo empiece a parpadear el LED "naranja", para seguidamente apagarse ambos y encenderse el LED "rojo". Seguidamente se debería de volver a empezar el mismo proceso otra vez. Incluso con la ayuda de un pulsador (de cuyo uso hablaremos en el siguiente apartado) se podría obligar a encender el LED "verde" (y apagar los demás), tal como ocurre en algunos semáforos reales.

Una vez vistos varios ejemplos donde se controlan dos o tres LEDs "en paralelo", aumentar el número de LEDs de nuestro circuito (conectando cada nuevo LED a una salida digital diferente) y escribir el sketch pertinente capaz de manejar esos cuatro, cinco, seis, siete LEDs o más es una tarea bastante sencilla. Un ejemplo

particular de esto es el caso de los displays 7-segmentos, los cuales, como ya vimos en el capítulo anterior, no son más que un conjunto de LEDs independientes (normalmente siete, de ahí el nombre) pero encapsulados físicamente de tal manera que al iluminarse un subconjunto de ellos se muestre una determinada cifra numérica (0, 1, 2, 3, 4, 5, 6, 7, 8 y 9) o algún carácter concreto. Como ejemplo de su funcionamiento presentamos el código siguiente, donde empleamos un display 7-segmentos como contador de segundos cíclico (es decir, un contador que empieza mostrando el "0", al cabo de un segundo el "1", al segundo siguiente el "2" y así hasta el "9", para a continuación volver a empezar por el "0" y seguir otra vez).

NOTA: Existen maneras más óptimas y compactas de codificar el comportamiento de un display 7-segmentos pero recurren a técnicas más avanzadas (como el uso de un array de bits para indicar las cifras a mostrar o, siguiendo la misma filosofía, el de un array de enteros combinado con valores booleanos) que no veremos.

Ejemplo 6.12

```
/*Figura que muestra la disposición de los LEDs dentro de un display 7-segmentos:
```

```

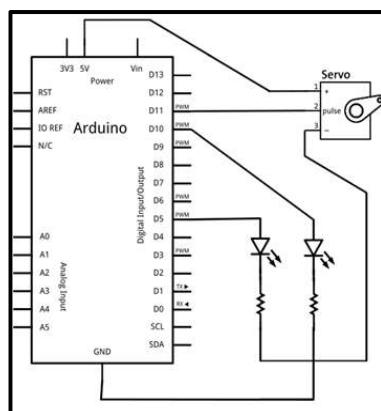
    _a_
    |   |
f|_g_|b
e|       |c
    |_d_|

*/
void setup() {
    pinMode(2, OUTPUT); //Pin conectado al LED 'a' del display 7-segmentos (mirar figura arriba)
    pinMode(3, OUTPUT); //Pin conectado al LED 'b' del display 7-segmentos (mirar figura arriba)
    pinMode(4, OUTPUT); //Pin conectado al LED 'c' del display 7-segmentos (mirar figura arriba)
    pinMode(5, OUTPUT); //Pin conectado al LED 'd' del display 7-segmentos (mirar figura arriba)
    pinMode(6, OUTPUT); //Pin conectado al LED 'e' del display 7-segmentos (mirar figura arriba)
    pinMode(7, OUTPUT); //Pin conectado al LED 'f' del display 7-segmentos (mirar figura arriba)
    pinMode(8, OUTPUT); //Pin conectado al LED 'g' del display 7-segmentos (mirar figura arriba)
}
void loop() {
    for (byte cifra = 0; cifra <= 9; cifra++) {
        switch (cifra) {
/*Se asignan los valores HIGH y LOW pertinentes a los parámetros de mostrar() para iluminar la cifra
adecuada. En este ejemplo, se ha asumido que el display 7-segmentos es de tipo ánodo común, lo que
significa que para que un segmento se ilumine, la salida correspondiente ha de estar en valor LOW; si
fuera de tipo cátodo común, el valor que haría iluminar el segmento sería HIGH. Notar que se puede
sustituir la constante LOW por su valor "real" (0). Igualmente, HIGH equivale a 1. */
            case 0: mostrar(0,0,0,0,0,1); break;
            case 1: mostrar(1,0,0,1,1,1,1); break;
        }
    }
}
```

EL MUNDO GENUINO-ARDUINO

```
case 2: mostrar(0,0,1,0,0,1,0); break;
case 3: mostrar(0,0,0,0,1,1,0); break;
case 4: mostrar(1,0,0,1,1,0,0); break;
case 5: mostrar(0,1,0,0,1,0,0); break;
case 6: mostrar(0,1,0,0,0,0,0); break;
case 7: mostrar(0,0,0,1,1,1,1); break;
case 8: mostrar(0,0,0,0,0,0,0); break;
case 9: mostrar(0,0,0,1,1,0,0); break;
}
delay(1000);           //Se muestra la cifra dada durante un segundo y...
mostrar(1,1,1,1,1,1,1); //...se apagan todos los LEDs antes de mostrar la siguiente cifra
}
//Una vez que se ha llegado al final de for (9), se vuelve a empezar (0) gracias al loop
}
void mostrar (boolean a, boolean b,boolean c,boolean d, boolean e, boolean f, boolean g){
    digitalWrite(2,a);
    digitalWrite(3,b);
    digitalWrite(4,c);
    digitalWrite(5,d);
    digitalWrite(6,e);
    digitalWrite(7,f);
    digitalWrite(8,g);
}
```

Hasta ahora hemos estudiado ejemplos de salidas digitales conectadas solamente a LEDs. Para finalizar este apartado daremos un paso más y estudiaremos el siguiente circuito, donde además de –otra vez– incorporar dos LEDs (conectados, como en los ejemplos anteriores, a las salidas digitales nº 5 y nº 10) añadiremos la presencia de un servomotor (conectado, en este caso, a la salida nº 11, tal como se puede ver en el diagrama siguiente):



El plan es hacer parpadear los dos LEDs de forma no bloqueante –igual que hicimos en el sketch 6.9/6.9Bis– para conseguir a la vez ir moviendo el eje del servomotor, movimiento que, a su vez, tampoco ha de ser bloqueante para que el parpadeo de los LEDs no "se congele". Es decir, la idea es controlar simultáneamente tres elementos independientes (2 LEDs y un servomotor) sin que el comportamiento de uno perjudique el desempeño de los otros. El siguiente código, en concreto, además de hacer parpadear de forma simultánea a los dos LEDs, hace girar de forma continua el eje del servomotor desde un ángulo de 10 grados hasta 170 y de ahí lo hace retroceder a la misma velocidad hasta 10 grados otra vez, para entonces volver a empezar de nuevo el giro (y así de manera ininterrumpida); es decir, hace lo mismo que ya conseguimos en el sketch 5.21 pero ahora evitando el uso de *delay()* mediante la técnica vista en el sketch 6.9Bis:

Ejemplo 6.13

```
#include <Servo.h>
Servo miservo1;
byte pinServo = 11;
byte pinLedA = 5;
byte pinLedB = 10;
void setup(){
    pinMode(pinLedA,OUTPUT);
    pinMode(pinLedB,OUTPUT);
    miservo1.attach(pinServo);
}
void loop() {
    maqEstFinLed1(1000);
    maqEstFinLed2(1000);
    maqEstFinServo1(10);
}
//-----Funciones idénticas al sketch 6.9Bis -----
void maqEstFinLed1(unsigned long intervalo){
    static boolean estadoLed = LOW;
    static unsigned long tUCambio = 0 ;
    unsigned long tActual = millis() ;
    if (tActual - tUCambio >= intervalo){
        tUCambio = tActual;
        if (estadoLed == LOW){ estadoLed = HIGH; }
        else { estadoLed = LOW; }
        digitalWrite(pinLedA,estadoLed);
    }
}
void maqEstFinLed2(unsigned long intervalo){
    static boolean estadoLed = LOW;
```

EL MUNDO GENUINO-ARDUINO

```
static unsigned long tUCambio = 0 ;
unsigned long tActual = millis() ;
if (tActual - tUCambio >= intervalo){
    tUCambio = tActual;
    if (estadoLed == LOW){ estadoLed = HIGH; }
    else { estadoLed = LOW; }
    digitalWrite(pinLedB, estadoLed);
}
//-----
/*El parámetro 'intervalo' indica el tiempo de espera entre giro y giro del eje del servo. Un valor mínimo para esta espera es necesario para dar tiempo al eje a alcanzar su nueva posición antes de hacer el giro siguiente.*/
void maqEstFinServo1(unsigned long intervalo){
    static byte posServo = 90;           //Variable de estado del servo (inicialmente en el centro)
    static unsigned long tUGiro = 0;      //Variable temporizadora del servo
    static int incr = 1;                /*Variable que indica el ángulo (en grados) a recorrer
por el eje del servo (a partir de la posición donde esté en ese momento) cuando se le ordena girar mediante miservo1.write(). Su valor debe poder tener valores negativos (para girar en sentido contrario). Se declara static porque su valor varía dentro del código de esta función y se ha de guardar entre llamadas (podría ser global, pero entonces todos los servos declarados tendrían un mismo comportamiento de giro).*/
    unsigned long tActual = millis();
    if (tActual - tUGiro > intervalo){      //Si es el momento de girar el eje...
        tUGiro = tActual;
        posServo = posServo + incr;          //...marco su próxima posición...
        /*Si esa posición marcada alcanza o excede de los límites de giro (en este ejemplo, 10 y
        170 grados), invierto el sentido del giro y vuelvo a marcar una nueva próxima posición
        (ya rectificada), descartando la anterior*/
        if ((posServo >= 170) || (posServo <= 10)) {
            //El cambio de signo de la variable 'incr' invierte la dirección de giro
            incr = -incr;
            //Marco —otra vez— la próxima posición, que ahora sí estará dentro del rango
válido
            posServo = posServo + incr;
        }
        miservo1.write(posServo); //...y giro el eje de 'miservo1' a esa posición marcada
    }
}
```

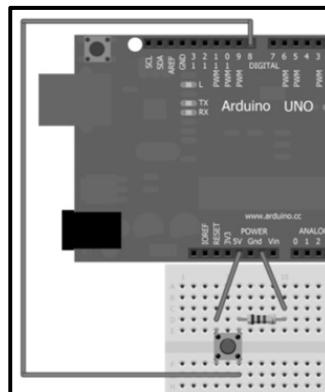
El código anterior puede resultar algo engorroso de manejar, y a la que añadamos más LEDs o servomotores, peor aún. Afortunadamente, al ser la mayoría de código bastante repetitivo, existe una manera de minimizar este inconveniente: emplear un par de conceptos de programación avanzados provenientes del lenguaje

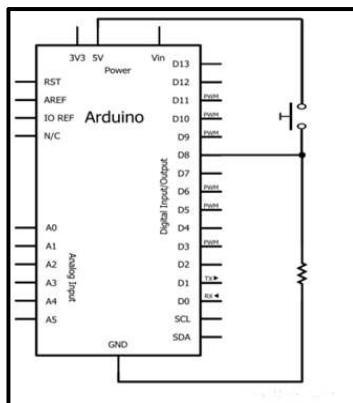
C++ (lenguaje sobre el cual está basado Arduino) llamados "clase" y "objeto". Este concepto permite al desarrollador escribir sus sketches de forma más lógica y clara porque ofrece la capacidad de agrupar las características y los comportamientos comunes de los diferentes "objetos" LEDs del circuito en una sola "clase" genérica –llamémosla clase *LED*– y los de todos los "objetos" servomotores en otra –llamémosla clase *servomotor*– (entre otros infinitos ejemplos). La gran ventaja de esto es que nos possibilita reutilizar gran parte del código ya escrito (precisamente el contenido dentro la "clase") cuando añadimos un nuevo LED o servomotor (es decir, un nuevo "objeto" perteneciente a una "clase" ya definida) al circuito. No obstante, los detalles de implementación necesarios para definir correctamente nuestras propias "clases" y "objetos" exigen un nivel de programación más avanzado que el nivel expuesto en este libro, por lo que esto no lo estudiaremos.

Ejemplos con entradas digitales (pulsadores)

En este apartado veremos varios ejemplos donde la placa Arduino realiza lecturas de señales digitales recibidas específicamente de parte de pulsadores. A priori esto puede parecer un poco extraño, ya que en principio un pulsador tan solo sirve para abrir o cerrar un circuito, pero hemos de saber que también tiene la capacidad de permitir la "monitorización" de su estado, de tal forma que con una señal de entrada digital la placa Arduino puede saber en todo momento si el pulsador está en posición abierta o posición cerrada. Para conseguir esta capacidad de "monitorización", los pulsadores han de estar obligatoriamente conectados a una resistencia "pull-up" o bien "pull-down" que evite fluctuaciones en esa señal de entrada. Así pues, tenemos dos posibilidades de circuitos, que pasamos a presentar:

- a) El dibujo siguiente (y su esquema asociado) se corresponde con el circuito diseñado para monitorizar el estado de un pulsador utilizando una resistencia "pull-down".





Podemos comprobar cómo las conexiones del pulsador son por un lado directa a la alimentación y por otro a tierra a través de la resistencia "pull-down" (pongamos que de $10K\Omega$). Existe un tercer cable, conectado entre el pulsador y la resistencia que va a parar a un pin digital de la placa Arduino (en este caso, el nº 8). Este pin digital deberá configurarse como pin de entrada porque allí será donde se reciba la señal que indique el estado del pulsador.

En esta configuración, cuando el botón esté abierto (es decir, cuando no esté pulsado), el "tercer cable" pasará a estar conectado a tierra a través de la resistencia "pull-down", por lo que recibirá una señal LOW. Cuando el botón esté cerrado (es decir, cuando sí esté pulsado), el "tercer cable" pasará a estar conectado directamente al pin de alimentación; aquí la clave está en observar que la mayor parte de corriente proveniente de ese pin de alimentación fluirá a través del camino más sencillo posible, que resulta ser el "tercer cable", en detrimento de la otra posibilidad (el camino a tierra a través de la resistencia "pull-down", el cual es mucho más costoso al paso de los electrones por existir precisamente esa resistencia). Esto hace que, al cerrar el botón, el "tercer cable" detecte una señal HIGH.

Una vez diseñado el circuito, ¿cómo podemos observar el estado actual del pulsador (obtenido en un sketch gracias a `digitalRead()`)? Una manera podría ser mediante el "Serial monitor". A continuación, mostramos un código de ejemplo que ilustra esto. Si lo ejecutamos, veremos que mientras tengamos pulsado el botón, en el "Serial monitor" aparecerá un 1, y cuando esté sin pulsar aparecerá un 0:

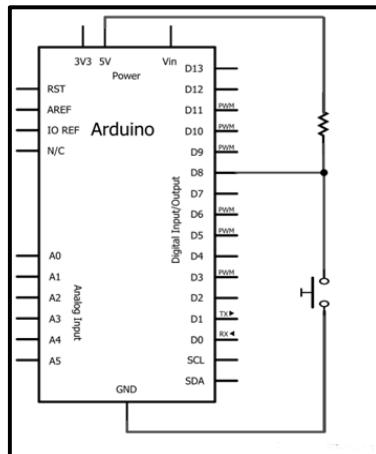
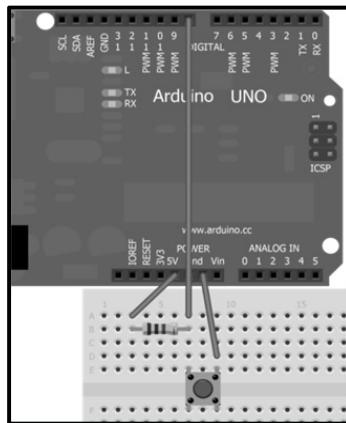
Ejemplo 6.14

```

int estadoBoton=0;           //Guardará el estado del botón (HIGH ó LOW)
void setup(){
    pinMode(8,INPUT); //Pin donde está conectado el pulsador
    Serial.begin(9600);
}
void loop() {
    estadoBoton=digitalRead(8);
    Serial.println(estadoBoton);
}

```

- b)** El dibujo siguiente (y su esquema asociado) se corresponde con el circuito diseñado para monitorizar el estado de un pulsador utilizando una resistencia "pull-up".



EL MUNDO GENUINO-ARDUINO

Aquí podemos comprobar cómo las conexiones del pulsador son por un lado a la alimentación a través de la resistencia "pull-up" (pongamos que de $10K\Omega$) y por otro directa a tierra. Existe un tercer cable, conectado entre el pulsador y la resistencia que va a parar a un pin digital de la placa Arduino (en este caso particular, el nº 8). Este pin digital deberá configurarse como pin de entrada porque allí será donde se reciba la señal que indique el estado del pulsador.

En esta configuración, cuando el botón esté abierto, el "tercer cable" estará conectado a la alimentación (a través de la resistencia "pull-up") por lo que recibirá de ella una señal HIGH. Cuando el botón esté cerrado, en cambio, la corriente proveniente del pin de la alimentación dispondrá de un camino muy sencillo por el que fluir (el que conecta directamente a tierra) por lo que muy poca cantidad de corriente atravesará el "tercer cable", lo que hará que este detecte una señal LOW. Es importante notar, por tanto, que en esta configuración con la resistencia "pull-up", se recibe LOW cuando se pulsa el botón y HIGH cuando se deja de pulsar, al contrario de lo "intuitivo" (que es lo que ocurre cuando se usan resistencias "pull-down").

El código que pusimos para monitorizar el pulsador en el circuito con resistencia pull-down sigue siendo válido ahora también con el circuito de resistencia "pull-up", pero veremos que los valores 0 y 1 están invertidos, tal como acabamos de comentar.

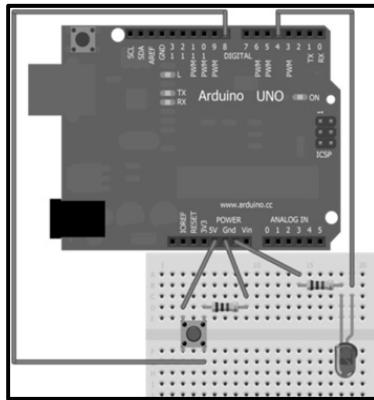
Recordemos que la placa Arduino tiene en cada uno de sus pines-hembra digitales una resistencia "pull-up" conectada internamente a la alimentación de 5V. Esto quiere decir que si para un determinado pin-hembra la activáramos (utilizando en la función *pinMode()*; la constante INPUT_PULLUP en vez de INPUT, recordemos), la conexión de un pulsador a ese pin-hembra sería mucho más sencilla. Concretamente tan solo deberíamos conectar un terminal del pulsador a tierra y el otro al pin-hembra deseado, y nada más. Eso sí, seguiríamos recibiendo una señal HIGH al tener el pulsador abierto y una señal LOW al tenerlo cerrado.

Implementación de pulsadores momentáneos

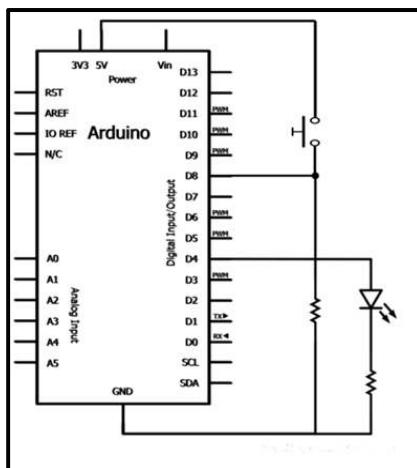
Una vez conocidas las dos configuraciones posibles (con resistencias "pull-up" o "pull-down"), podemos empezar ya a diseñar circuitos que sean capaces de detectar el estado actual de un pulsador y reaccionar en consecuencia. Empezaremos por un caso muy claro y directo: el encendido de un LED mientras se mantiene pulsado un botón. Para ello en realidad lo que tenemos que hacer es diseñar dos circuitos independientes: uno para el manejo del pulsador y otro para el encendido

CAPÍTULO 6: ENTRADAS Y SALIDAS

del LED. El primero lo acabamos de estudiar (elegiremos el de la configuración "pull-down") y el segundo no es más que el primer circuito que vimos en los ejemplos de las salidas digitales. Los dos juntos tienen un aspecto como el mostrado en la siguiente figura:



En el dibujo anterior se puede observar que hay dos cables conectados a tierra, cada uno perteneciente a uno de los dos circuitos independientes. Esto se ha hecho así por claridad, pero es más habitual unir todos los cables a tierra físicamente en un solo cable final. En todo caso, la tierra siempre ha de ser común (cosa que en este caso está garantizado porque las diferentes tierras ofrecidas por la placa Arduino están todas conectadas entre sí, tal como se aprecia en el esquema del circuito).



En el esquema eléctrico anterior se ve más claro aún que este circuito no es más que la unión de dos circuitos independientes ya vistos anteriormente. Así pues, aparentemente, cada uno de estos dos circuitos no está relacionado con el otro, pero

EL MUNDO GENUINO-ARDUINO

aquí es cuando interviene nuestro sketch, el cual funcionará como un "pegamento" entre las dos partes. Nuestro programa irá leyendo el estado del pulsador constantemente y cuando detecte que este esté siendo pulsado (al recibir una señal HIGH por el pin de entrada nº 8), reaccionará consecuentemente enviando una señal HIGH por el pin de salida nº 4 para encender el LED.

Ejemplo 6.15

```
boolean estadoBoton=0;           //Guardará el estado del botón (HIGH ó LOW)
void setup(){
    pinMode(4,OUTPUT); //Donde está conectado el LED
    pinMode(8,INPUT); //Donde está conectado el pulsador
}
void loop() {
    estadoBoton=digitalRead(8);
    //Si se detecta que el botón está pulsado, se enciende el LED
    if (estadoBoton == HIGH) {
        digitalWrite(4,HIGH);
    }
    //Si no, no
    } else {
        digitalWrite(4,LOW);
    }
}
```

Con el mismo circuito, y teniendo en cuenta el comportamiento del código anterior, debería ser bastante sencillo para el lector adivinar qué es lo que hará el siguiente sketch:

Ejemplo 6.16

```
byte pinLed = 4; //Salida digital donde está conectado el LED.
byte pinBoton = 8; //Entrada digital donde está conectado el pulsador
/*Se ha definido las dos variables globales anteriores para, en caso de cambiar los pines-hembra de Arduino donde se conectan el LED y/o el pulsador, tan solo haya que indicar el valor numérico de esos nuevos pines-hembra en las líneas anteriores y en ningún sitio más del código */
boolean estadoBoton=0;
void setup(){
    pinMode(pinLed,OUTPUT); //Donde está conectado el LED
    pinMode(pinBoton,INPUT); //Donde está conectado el pulsador
}
void loop() {
    estadoBoton=digitalRead(pinBoton);
    if (estadoBoton == HIGH) {
        parpadeo(pinLed);
    }
}
```

```

        digitalWrite(pinLed,LOW);
    }
}

void parpadeo(byte led){
    digitalWrite(led, HIGH);
    delay(100);
    digitalWrite(led, LOW);
    delay(100);
}

```

También podemos optar por un encendido "retardado": el siguiente sketch enciende el LED solamente tras haber estado pulsando ininterrumpidamente el pulsador durante 3 segundos, no antes (y al dejarlo de pulsar lo apaga inmediatamente):

Ejemplo 6.17

```

byte pinLed = 4;
byte pinBoton = 8;
boolean estadoBoton=0;
boolean yaPulsado = false;
unsigned long inicio;
void setup(){
    pinMode(pinLed,OUTPUT);
    pinMode(pinBoton,INPUT);
}
void loop() {
    estadoBoton=digitalRead(pinBoton);
    //Si el botón está pulsado pero no lo estaba ya de antes, inicio el contador...
    if (estadoBoton == HIGH && yaPulsado == false) {
        inicio = millis();
        yaPulsado = true; //...y marco que el botón ya ha sido pulsado
    }
    //Si el botón está marcado como pulsado y de eso hace 3 segundos o más...
    if (yaPulsado == true && (millis() - inicio >= 3000 )) {
        digitalWrite(pinLed,HIGH); //...enciendo el LED
    }
    /*En el instante que se deje de pulsar el botón, apago el LED
     *y reseleo la marca de haber sido pulsado. */
    if (estadoBoton == LOW) {
        digitalWrite(pinLed,LOW);
        yaPulsado = false;
    }
}

```

EL MUNDO GENUINO-ARDUINO

Otra modificación del proyecto anterior (hay tantas como la imaginación nos permita) podría ser añadir al circuito un nuevo LED (junto con su correspondiente divisor de tensión) conectado al pin digital nº 5 (por ejemplo) y modificar el código 6.15 para que cuando se mantuviera pulsado el botón se encendiera un LED y cuando se soltara se encendiera el otro. El truco está en enviar una señal HIGH a un LED y una señal LOW al otro al mismo tiempo, según el estado detectado del pulsador. Se deja también como ejercicio al lector.

Por otro lado, recordar que si hubiéramos querido realizar los ejemplos anteriores usando la versión de pulsador con resistencia "pull-up", deberíamos haber tenido en cuenta que si *digitalRead()* devuelve LOW, es cuando el botón está pulsado y cuando devuelve HIGH es cuando el botón no está activado. Nada más.

Implementación de pulsadores mantenidos

En este apartado transformaremos el botón momentáneo que ya tenemos en un botón mantenido; es decir, conseguiremos que el botón no tenga que aguantarse pulsado para activar una salida, sino que pulsándolo una vez ya la active y pulsándolo otra vez la desactive. Pensemos, por ejemplo, en el botón de encendido/apagado de un mando a distancia del televisor: sería muy pesado tener que mantenerlo pulsado todo el rato para hacer funcionar el aparato. Como ejemplo utilizaremos el mismo circuito de los códigos anteriores, con un pulsador conectado a una resistencia "pull-down" y a la entrada digital número 8 por un lado y con un LED conectado a un divisor de tensión y la salida digital número 4 por otro. El código, presentado a continuación (y explicado en el párrafo siguiente), es lo que cambia:

Ejemplo 6.18

```
byte pinLed = 4;
byte pinBoton = 8;
boolean estadoActual=0;           //Guarda el estado actual del botón
boolean estadoUltimo=0;           //Guarda el último estado del botón
int contador=0;                  //Guarda las veces que se pulsa el botón
void setup(){
    pinMode(pinLed,OUTPUT);
    pinMode(pinBoton,INPUT);
    Serial.begin(9600);
}
void loop(){
    //Leo el nuevo estado actual del botón
    estadoActual=digitalRead(pinBoton);
    //Si este cambia respecto el estado justo anterior...
    if (estadoActual != estadoUltimo){
```

```

/*...lo notifico. Pero para ello tengo que comprobar que el cambio sea una
pulsación y no una liberación (es decir, que el estado actual sea HIGH (si
hubiéramos usado una resistencia pull-up, el valor a comprobar de "estadoActual"
entonces sería LOW) */
if (estadoActual == HIGH) {
    //Si es una pulsación, aumento el contador de pulsaciones
    contador = contador + 1;
    /*Notar que se ha de poner varios Serial.print para intercalar en una sola
    sentencia frase literal con valores de variables*/
    Serial.print ("Esta es la pulsación nº ");
    Serial.println(contador);
}
//Guardo el estado actual para la siguiente comprobación (en el próximo loop())
estadoUltimo= estadoActual;
/*Cambiamos el estado del LED según el valor del contador de pulsaciones teniendo en
cuenta que si pulsación sirve para activar la salida digital (esto es, encender el LED), la
siguiente pulsación sirve para desactivarla, y así. Esto significa que, si partimos inicialmente
de contador=0 y la salida digital a LOW, un número impar del contador indicará las
pulsaciones destinadas a encender el LED y un número par indicará las pulsaciones que lo
apagan. Así pues: */
if (contador % 2 == 0 ) {
    digitalWrite(pinLed, LOW);
} else {
    digitalWrite(pinLed, HIGH);
}
}

```

El sketch anterior continuamente está monitorizando el estado del botón. Si detecta en algún momento que el botón sufre un cambio (es decir, si sufre una pulsación –pasa de no estar apretado a sí estarlo– o bien sufre una liberación –pasa de estar apretado a no estarlo–), se comprueba de cuál de estos dos cambios se trata (pulsación o liberación). Si es el segundo caso, no hará nada, pero si se trata de una pulsación, se aumenta en uno el valor del contador de pulsaciones y se envía ese valor nuevo al canal serie para notificar el hecho. Este contador será importante después, más allá de que ahora sea un simple numerito mostrado en el "Serial monitor". Antes de proceder con el encendido o apagado del LED, el sketch aún tiene que guardar el estado actual del botón (haya cambiado o no: por eso la línea correspondiente está fuera de cualquier *if*) para que en la nueva repetición del *loop()* se compare este estado con el nuevo que tendrá en esa siguiente repetición. Y así constantemente.

EL MUNDO GENUINO-ARDUINO

Para mantener el LED encendido (o apagado, según el caso) sin necesidad de mantener pulsado todo el rato el botón, se emplea el contador comentado antes. La clave está en darse cuenta de que las liberaciones (es decir, los cambios de estado HIGH a LOW –si usamos "pull-down"—) son ignorados por completo, ya que lo único que se cuenta (mediante la variable *contador*) son las pulsaciones (es decir, los cambios de estado LOW a HIGH –si usamos "pull-down"—). Pero además, no todas las pulsaciones son iguales, porque mientras una pulsación debe servir para encender el LED, la siguiente debe servir para apagarlo, y la siguiente para volverlo a encender, etc. Es decir, suponiendo que el LED está apagado al iniciar el sketch, cuando el botón se pulse una vez ("vez nº 1") querremos que el LED se encienda, cuando se pulse la siguiente vez ("vez nº 2") se apague, cuando se pulse la siguiente vez ("vez nº 3") se vuelva a encender, y así sucesivamente. Podemos fijarnos que aquí se cumple un patrón: si la pulsación ocurre una vez impar, el LED se encenderá y si la vez es par, se apagará. Por tanto, la idea es comprobar que el contador tenga un valor par o impar. A medida que se realicen pulsaciones este contador irá aumentando hasta llegar a su límite máximo marcado por su tipo de datos, pero en ese momento su valor se reseteará por debajo y seguirá aumentando, así que en este sentido no habrá ningún problema.

¿Y cómo se sabe si un número es par o impar? Dividiéndolo entre dos y observando el resto de la división: si este es 0, el número es par. El lenguaje Arduino consta de un operador matemático (llamado "módulo" y representado con el signo %) que permite obtener precisamente el resto de una división. Así que ya lo tenemos todo.

El código anterior se puede reescribir de una manera más ordenada y clara si la parte relacionada con el control de las pulsaciones se convierte en una función propia. De esta forma, dicha parte de control podría ser reutilizada en otros proyectos independientes de forma sencilla y práctica. Así pues, si hacemos esto, el código anterior quedará de la siguiente manera (hemos suprimido las líneas relacionadas con la comunicación serie, ya que no aportaban nada nuevo):

Ejemplo 6.19

```
byte pinLed = 4;  
byte pinBoton = 8;  
void setup(){  
    pinMode(pinLed,OUTPUT);  
    pinMode(pinBoton,INPUT);  
}  
void loop(){  
    /*La función propia 'pulsacionEncendido()' devuelve 'true' si la pulsación recibida por el
```

```

botón conectado a la entrada digital indicada como parámetro se corresponde con una
pulsación de encendido del LED (o 'false' en todos los demás casos). */
boolean pulE=pulsacionEncendido(pinBoton);
if (pulE == true){
    digitalWrite(pinLed, HIGH);
} else {
    digitalWrite(pinLed, LOW);
}
boolean pulsacionEncendido(byte pin) {
    //Variables de estado del pulsador (si solo hay uno, pueden ser globales)
    static boolean estadoUltimo=0;
    static int contador=0;
    //Código de detección de la pulsación (o liberación)
    boolean estadoActual=digitalRead(pin);
    if (estadoActual != estadoUltimo){
        if (estadoActual == HIGH) { //Si es una pulsación...
            contador = contador + 1;
        }
    }
    estadoUltimo= estadoActual;
    /*Código de distinción entre pulsación para encender (contador impar)
     o para apagar (contador par)*/
    if (contador % 2 == 0 ) { return false; }
    else { return true; }
}

```

¿Cómo se podría modificar el código anterior para que mostrara por el "Serial monitor" una cuenta atrás de 10 pulsaciones (por ejemplo) y que al llegar a 0 (es decir, al hacer diez pulsaciones) se imprimiera un mensaje final? Pista: el truco está en usar un valor inicial para la variable *contador* igual a 10 y modificar el interior de la sección *if (estadoActual == HIGH) {}* con la introducción allí de dos nuevos *if*: uno para comprobar si esa variable es aún mayor que 0 (en cuyo caso, se disminuiría el valor de *contador* y se mostraría la correspondiente cuenta atrás), y otro para comprobar si esa variable es exactamente igual que 0 (mostrando entonces el mensaje final). ¿Se seguiría encendiendo el LED una vez mostrado el mensaje final? Se deja como ejercicio.

Los pulsadores momentáneos no solamente pueden provocar dos cambios en una salida digital ("encender" y "apagar"), sino que también pueden trabajar en varios "modos", provocando en cada uno de ellos un cambio diferente ("encender", "apagar", "parpadear a una velocidad", "parpadear a otra velocidad", etc.). Lo más sencillo es que la activación de esos modos sea de tipo secuencial; esto es: que con

EL MUNDO GENUINO-ARDUINO

una pulsación se entre en un modo, con la siguiente pulsación se entre en otro, con la siguiente en otro... y así hasta volver otra vez al primer modo. El siguiente sketch consigue precisamente esto: cambiar el modo de funcionamiento del circuito (que en este caso significa cambiar el intervalo de parpadeo del LED) mediante sucesivas pulsaciones de un botón. Concretamente, en el sketch se definen tres modos: el modo "0" (que provoca el parpadeo del LED cada 100 ms), el modo "1" (que cambia el periodo del parpadeo a 200ms) y el modo "2" (que lo cambia a 500ms), activándose cada uno de ellos con una pulsación (y volviendo con otra del modo "2" al "0" para volver a empezar). Para implementar el parpadeo del LED se ha utilizado la función *cambioLed()* introducida en varios ejemplos del apartado anterior debido a su naturaleza no bloqueante, lo que permite al sketch estar pendiente de las eventuales pulsaciones del botón mientras se va realizando el parpadeo:

Ejemplo 6.20

```
byte pinLed = 4;
byte pinBoton = 8;
unsigned long tActual;
void setup(){
    pinMode(pinLed,OUTPUT);
    pinMode(pinBoton,INPUT);
}
void loop(){
    tActual = millis();
    /*La función pulCambioModo() devuelve el periodo del parpadeo del LED, el cual depende
    del modo de trabajo actual. Ese modo se puede cambiar pulsando sucesivamente al botón */
    unsigned long intervalo = pulCambioModo(pinBoton);
    cambioLed(pinLed, intervalo); //Ilumino el LED con el periodo obtenido en la línea anterior
}
unsigned long pulCambioModo(byte pin){
    static boolean estadoUltimo=0; //Variable usada para gestionar el modo mantenido del botón
    static byte modo = 0;           //Variable que guarda el modo de trabajo actual del pulsador
    static unsigned long valorRetorno = 100; /*Es importante que valorRetorno sea static para
    mantener en las sucesivas iteraciones del loop() el periodo de parpadeo seleccionado, aunque el botón
    haya sido liberado. (si no, en cada llamada a pulCambioModo() se resetearía a su valor inicial, el cual,
    por cierto, es el correspondiente al 'modo 0'*/
    boolean estadoActual=digitalRead(pin); //Leo el estado del pulsador.
    if (estadoActual != estadoUltimo){      //Si hay cambio...
        if (estadoActual == HIGH) { //...confirmo que sea una pulsación (y no una liberación)
            modo = (modo + 1) % 3; //Si es así, activo el modo de trabajo pertinente
            switch(modo){       //Según el modo activado, defino un valor de retorno diferente
                case 0: valorRetorno=100; break;
                case 1: valorRetorno=200; break;
                case 2: valorRetorno=500; break;
            }
        }
    }
}
```

```

        }
    }
    estadoUltimo= estadoActual; //Guardo el estado actual para la siguiente comprobación
    return valorRetorno; //Devuelvo el valor definido en el switch según el modo activado
}
void cambioLed(byte led, unsigned long intervalo){
    static boolean estadoLed = 0;
    static unsigned long tUCambioLed;
    if (tActual - tUCambioLed >= intervalo){
        tUCambioLed = tActual;
        if (estadoLed == LOW){ estadoLed = HIGH; }
        else { estadoLed = LOW; }
        digitalWrite(led, estadoLed);
    }
}

```

Tal vez la línea más interesante del código anterior sea la que, tras haberse detectado una pulsación, cambia el modo de trabajo del circuito, es decir, la línea: *modo = (modo + 1) % 3.* ¿Qué hace exactamente? Fijarse que se trata de una suma a la que se le aplica la operación "módulo 3" (es decir, una suma cuyo resultado se divide entre 3 obteniéndose el resto de esa división). Así pues, si inicialmente la variable *modo* vale 0, su nuevo valor tras realizar la operación $(0+1)\%3=1\%3$ será 1; a la siguiente ejecución de esa misma línea, el nuevo valor de *modo* será entonces, por tanto, $(1+1)\%3=2\%3=2$, y en la siguiente, $(2+1)\%3=3\%3=0$, volviendo a empezar el ciclo otra vez. Es decir, aunque vayamos sumando 1 infinitas veces al valor actual de *modo*, este no saldrá del ciclo 0, 1 y 2 una y otra vez, y además de forma secuencial. Esta es una de las aplicaciones más importantes de la operación "módulo x": forzar a que los resultados de un determinado cálculo (en este caso, una suma) se encuentren "encerrados" de forma cíclica dentro de un rango de valores dados (concretamente desde el valor 0 hasta x-1).

Otro proyecto que no debería ser demasiado difícil para el lector (una vez estudiado el ejemplo anterior) es la sustitución en el circuito del LED por un display de 7-segmentos y el desarrollo del código necesario para controlar mediante la activación momentánea de un pulsador la visualización de cifras en ese display (por ejemplo, haciendo que cada vez que se detecte una pulsación momentánea, vaya aumentando la cifra mostrada –de 0 a 1, de 1 a 2...– hasta llegar al 9 y entonces volver a empezar desde cero). Se deja como ejercicio.

Evitando el rebote ("bounce") en los pulsadores

Desgraciadamente, es posible que al probar el sketch 6.14 (por mencionar un sketch cualquiera de los previamente estudiados en este apartado) se perciba que "a veces" una sola pulsación del botón genera más de un mensaje por el "Serial monitor" o que, en cambio, al ejecutar el sketch 6.15 (por poner otro caso de entre los vistos) el LED "a veces" no reaccione a una pulsación concreta. Estos dos problemas aparentemente tan distintos están, sin embargo, relacionados: ambos ocurren porque durante el primer milisegundo de cada presionado (y soltado) del botón se producen a nivel electrónico pequeñas variaciones de la señal de entrada que hacen que los valores HIGH y LOW obtenidos del pulsador alternen rápidamente hasta que no se stabilizan en el valor adecuado.

Este fenómeno se llama "bounce" (rebote) y es inevitable por la propia construcción mecánica de este tipo de pulsadores: cuando el botón se aprieta, una lámina existente bajo este es presionada y hace contacto con dos extremos conductores; cuando el botón se deja de apretar, esta lámina retorna. Pero durante el primer milisegundo de la pulsación, esta lámina puede rebotar varias veces entre sus dos posiciones hasta que queda fijada finalmente en su posición correcta, provocando por tanto lecturas alternas del estado del pulsador. Esto causa que a veces nuestro sketch no recoja el valor correcto del estado del pulsador y parezca estar recibiendo múltiples pulsaciones y liberaciones ficticias, y por tanto, múltiples valores HIGH y LOW sin sentido.

Este problema puede ser solucionado por dos vías distintas: por "hardware" o por "software". La primera implica acompañar al pulsador (ya sea de tipo "pull-down" o "pull-up", esto es igual) de un circuito eléctrico suplementario que se encargue de amortiguar la señal rebotada. Existen muchos diseños posibles para este circuito amortiguador (llamado también circuito "debouncer"): desde los más sofisticados (donde se emplean componentes electrónicos especializados en esta tarea específica) hasta los más "artesanales" (formados tan solo por un condensador, una resistencia y, opcionalmente, un diodo). Sin embargo, lo más sencillo es optar por la solución "software", que básicamente consiste en modificar nuestro sketch para que a la hora de obtener el estado de un pulsador (si HIGH o LOW) realice dos lecturas con una diferencia de unos milisegundos entre ellas. Si en estas dos lecturas obtenemos valores diferentes, significa que estamos en un momento de rebote y por tanto nuestro sketch no debe hacer nada hasta que vuelva a comprobar si la señal ya está stabilizada; si las dos lecturas son la misma, significa que el estado del botón finalmente es "el que ha de ser" (es decir, no lo hemos pillado por un rebote de casualidad) y nuestro sketch podrá tomar por tanto dicho valor como bueno. En la

práctica, dentro de nuestro código esta solución "software" se traduce en sustituir cada una de las líneas similares a:

```
valor = digitalRead(pinBoton);
//...a partir de aquí se comprueba 'valor' y se actúa en consecuencia
```

por:

```
valor1 = digitalRead(pinBoton); //Primera lectura del estado del pulsador
delay(10); //10ms es un tiempo razonable entre lectura y lectura
valor2 = digitalRead(pinBoton); //Segunda lectura del estado del pulsador
/*Si las dos lecturas no son iguales, estamos en un periodo de "rebote". Por tanto, no se hace nada y se vuelve a probar a la siguiente iteración del loop. Si entonces sí son iguales, ya estamos en un periodo estable y, por tanto, damos por buenas las lecturas*/
if (valor1 == valor2) {
    //...a partir de aquí se comprueba 'valor1' (o 'valor2', tanto da) y se actúa en consecuencia
}
```

El trozo de código anterior podría ser utilizado como plantilla en todo aquel lugar dentro de nuestro sketch donde sea necesario estar pendiente del estado de un pulsador. Por ejemplo, si lo introducimos en el sketch 6.15 (el cual implementa, recordemos, un pulsador "pull-down" momentáneo que al estar presionado enciende un LED), el resultado global será este (el trozo añadido aparece en negrita):

Ejemplo 6.21

```
byte pinLed = 4;
byte pinBoton = 8;
void setup(){
    pinMode(pinLed,OUTPUT);
    pinMode(pinBoton,INPUT);
}
void loop() {
    boolean estadoBoton1=digitalRead(pinBoton);
    delay(10);
    boolean estadoBoton2=digitalRead(pinBoton);
    if (estadoBoton1 == estadoBoton2) {
        if (estadoBoton1 == HIGH) { digitalWrite(pinLed,HIGH); }
        else { digitalWrite(pinLed,LOW); }
    }
}
```

Si se ejecuta el código anterior, se puede comprobar que el circuito funciona exactamente igual que lo hacía con el código 6.15 pero de una forma mucho más

EL MUNDO GENUINO-ARDUINO

fiable y sensible, reaccionando tal y como se espera de un pulsador, sin comportamientos extraños. El único inconveniente de esta técnica es que, al utilizar *delay()* entre lectura y lectura, nuestro sketch está siendo bloqueado durante ese (pequeño) tiempo. Por lo general, esto no debería ser un problema, pero si quisieramos evitar este bloqueo, una opción sería emplear la técnica estudiada en el apartado anterior; esto es: ir comprobando a cada iteración del *loop()* –con la ayuda de un contador definido mediante *millis()*– si ya se ha superado el intervalo de tiempo pertinente entre una primera lectura (previamente guardada) y una segunda lectura del pulsador, momento en el que se haría la comparación entre ambas. No obstante, esta técnica es algo engorrosa de programar, por lo que lo más sencillo en estos casos es emplear alguna de las librerías de terceros especialmente diseñadas para gestionar de forma no bloqueante (y mucho más intuitiva) el "bounce" de los pulsadores. Ejemplos de algunas de estas librerías (en ningún orden en particular) son:

Bounce2 (<https://github.com/thomasfredericks/Bounce2>)

Tigger (<https://github.com/GreyGnome/Tigger>)

ButtonReader (https://github.com/ElectricRCAircraftGuy/eRCaGuy_ButtonReader)

SwitchManager (<http://gammon.com.au/Arduino/SwitchManager.zip>)

DebounceInput (<https://github.com/MajenkoLibraries/DebouncedInput>)

ButtonV2 (<https://github.com/AndrewMascolo/ButtonV2>) Esta librería es más general ya que, además de proveer métodos "antirrebote", también permite implementar pulsadores momentáneos de una forma muy sencilla, establecer diferentes modos de trabajo, contar el tiempo de pulsación, etc.

Juegos

A continuación pondremos en práctica todo lo estudiado hasta ahora sobre pulsadores diseñando varios juegos donde la habilidad para pulsar a tiempo el botón será fundamental. El primer juego consiste en un circuito formado por tres LEDs conectados cada uno (a través de su respectivo divisor de tensión en serie) a un pin-hembra digital diferente de la placa Arduino (concretamente, en nuestro sketch son los pines-hembra 5, 6 y 7) y un pulsador, conectado a otro pin digital (el nº 8). Los LEDs se irán encendiendo y apagando de forma cíclica y secuencial (es decir, siguiendo el orden 5, 6, 7, 6, 5, 6, 7, 6, 5, 6, 7, etc.) y cuando el LED del medio (el 6) se encienda, el jugador deberá apretar el pulsador. Si acierta, se mostrará un mensaje por el "Serial monitor" y la velocidad de la secuencia de iluminación de los LEDs aumentará (y también lo hará por tanto la dificultad). El tiempo inicial entre encendido y encendido de los LEDs es 200ms, pero a cada acierto este valor disminuirá en 20ms, hasta llegar a un tiempo entre encendidos de 10ms, momento en el cual se volverá al tiempo inicial de 200ms.

Ejemplo 6.22

```

int leds[]={5,6,7};
int i=0;
int tiempo=200;
void setup () {
    for(i=0;i<=2;i++) {
        pinMode(leds[i],OUTPUT);
    }
    pinMode(8,INPUT);
    Serial.begin(9600);
}
void loop () {
    //Recorro los LEDs del array, iluminándolos y apagándolos
    for(i=0;i<=2;i++) {
        digitalWrite(leds[i],HIGH);
        delay(tiempo);
        //Antes de apagar cada LED, miro si el jugador ha acertado
        compruebaAcierto();
        digitalWrite(leds[i],LOW);
        delay(tiempo);
    }
}
void compruebaAcierto(){
    /*Si se tiene pulsado el botón y ahora mismo el LED encendido es el que ocupa
    la posición 1 dentro del array (es decir, el del medio), se acertó */
    if(digitalRead(8)==HIGH && i==1) {
        Serial.println("Acierto");
        tiempo=tiempo-20;
        if(tiempo<10){
            tiempo=200;
        }
    }
}
}

```

Otro juego curioso es el siguiente, se trata de implementar el juego de "los trileros", en el cual tenemos también tres LEDs que durante un breve lapso de tiempo se iluminan en una secuencia rápida y aleatoria. El usuario deberá adivinar cuál de los tres LEDs es el último en iluminarse apretando el pulsador correspondiente. Existe un pulsador por cada LED, y en el código se han configurado con las resistencias "pull-up" internas de la placa Arduino (por lo que su conexión no requiere ninguna resistencia externa, tal como ya se ha comentado en párrafos anteriores). Si el usuario acierta, se enviará un mensaje de felicitación por el canal serie; si no, se enviará un mensaje de consuelo.

EL MUNDO GENUINO-ARDUINO

Ejemplo 6.23

```
byte LEDizq = 2; //Pin-hembra donde está conectado el LED izquierdo
byte LEDmedio = 3; //Pin-hembra donde está conectado el LED del medio
byte LEDder = 4; //Pin-hembra donde está conectado el LED derecho
byte BotonIzq = 7; //Pin-hembra donde está conectado el pulsador izq.
byte BotonMed = 8; //Pin-hembra donde está conectado el pulsador med.
byte BotonDer = 9; //Pin-hembra donde está conectado el pulsador der.
byte LEDrandom;
byte LEDultimo;
byte respuesta;
void setup() {
    pinMode(LEDizq, OUTPUT);
    pinMode(LEDmedio, OUTPUT);
    pinMode(LEDder, OUTPUT);
    pinMode(BotonIzq, INPUT_PULLUP);
    pinMode(BotonMedio, INPUT_PULLUP);
    pinMode(BotonDer, INPUT_PULLUP);
    Serial.begin(9600);
}
void loop() {
    //Uso como semilla aleatoria el ruido leído en una entrada analógica (ver siguiente apartado)
    randomSeed(analogRead(0));
    //Enciendo los tres LEDs de forma aleatoria
    for(byte x = 1; x <= 9; x++) {
        LEDrandom = random(2,5);
        digitalWrite(LEDrandom, HIGH); delay(15);
        digitalWrite(LEDrandom, LOW); delay(10);
        if(x == 9) { LEDultimo = LEDrandom; }
    }
    /*Me espero mientras no se reciba ninguna respuesta. La condición del while siempre es cierta
    (se trata de un bucle infinito), pero cuando se detecte una pulsación, se sale de él y se continúa
    el programa */
    while(1 == 1) {
        if(digitalRead(BotonIzq) == LOW) { respuesta = 2; break; }
        else if(digitalRead(BotonMedio) == LOW) { respuesta = 3; break; }
        else if(digitalRead(BotonDer) == LOW) { respuesta = 4; break; }
    }
    //Compruebo si la respuesta es correcta o no
    if(respuesta == LEDultimo) { Serial.print("Muy bien"); }
    else { Serial.print("Vaya, lo siento"); }
}
```

Con el mismo circuito del ejemplo anterior (tres LEDs y tres pulsadores –en este caso "pull-up"–) podemos implementar un grabador/reproductor de secuencias de encendidos de LEDs. Concretamente, el sketch siguiente está dividido en dos partes: la primera registra la secuencia de iluminaciones de LEDs producida por las diferentes pulsaciones realizadas indistintamente a los tres botones del circuito (cada uno ilumina un determinado LED) hasta llegar al número máximo permitido de pulsaciones, momento en el cual el sketch procede a iluminar automáticamente los LEDs siguiendo la secuencia recién registrada. Una vez hecho esto, el sketch se dispone otra vez a atender nuevas pulsaciones para grabar una nueva serie de iluminaciones (borrando la serie anterior).

Ejemplo 6.24

```
/*Array bidimensional que relaciona los pines donde están conectados
los LEDs (2,3 y 4) con sus respectivos pulsadores (7,8 y 9) */
byte ledsPuls[3][3]={ {2,7},{3,8},{4,9} };
/*Array donde se guardarán uno tras otro los nº de pin donde están conectados los LEDs que se van
encendiendo en el proceso de grabación para posteriormente poder reproducir la misma secuencia */
byte secuencia[]={0,0,0,0,0,0,0,0,0};
//Variable que marca la posición del último pin guardado hasta el momento dentro de secuencia[]
byte s=0;
//Nº máximo de pulsaciones que se guardarán (como máximo será el nº de elementos de secuencia[])
byte nmaxpuls=10;
void setup () {
    for(byte i=0;i<=2;i++){
        /*Defino qué pines son las entradas (los LEDs: ledsPuls[0][0], ledsPuls[1][0] y ledsPuls[2][0])
        y qué pines son las salidas (los pulsadores: ledsPuls[0][1], ledsPuls[1][1] y ledsPuls[2][1]) */
        pinMode(ledsPuls[i][0],OUTPUT);
        pinMode(ledsPuls[i][1],INPUT_PULLUP);
    }
}
void loop(){
    guardaSec();           //Cuando reciba una pulsación, la guarda, pero...
    if(s==nmaxpuls){      //...si se ha llegado al límite de pulsaciones
        reproduceSec();   //...entonces reproduce la secuencia guardada...
        s=0;              //...y reinicia la variable 's' para volver a empezar una nueva grabación
    }
}
/*Almaceno en el array secuencia[] los nº de pin correspondientes
a los LEDs iluminados por las pulsaciones realizadas */
void guardaSec(){
    if (digitalRead(ledsPuls[0][1]) == LOW) { //Si el botón '[0][1]' se ha pulsado (es "pull-up")...
        secuencia[s]=ledsPuls[0][0];          //...grabo en secuencia[s] su LED asociado...
        digitalWrite(ledsPuls[0][0],HIGH);     //...e ilumino ese LED para que se vea
    }
}
```

EL MUNDO GENUINO-ARDUINO

```
delay(200);
digitalWrite(ledsPuls[0][0],LOW);
s++; //Me muevo al siguiente elemento de secuencia[] (para ubicar allí la próxima grabación)
}
if (digitalRead(ledsPuls[1][1]) == LOW) {
    secuencia[s]=ledsPuls[1][0];
    digitalWrite(ledsPuls[1][0],HIGH);
    delay(200);
    digitalWrite(ledsPuls[1][0],LOW);
    s++;
}
if (digitalRead(ledsPuls[2][1]) == LOW) {
    secuencia[s]=ledsPuls[2][0];
    digitalWrite(ledsPuls[2][0],HIGH);
    delay(200);
    digitalWrite(ledsPuls[2][0],LOW);
    s++;
}
}
//Función que ilumina los LEDs según la secuencia previamente almacenada
void reproduceSec(){
    for(byte i=0;i<=nmaxpuls;i++){
        digitalWrite(secuencia[i],HIGH);
        delay(200);
        digitalWrite(secuencia[i],LOW);
        delay(200);
    }
}
```

Evidentemente, con pulsadores no solo podemos controlar LEDs, sino cualquier otro tipo de actuador. Por ejemplo, en un circuito con dos pulsadores conectados a los pines de entrada digital nº 7 y nº 8, respectivamente (además de a la alimentación y a tierra a través de una resistencia "pull-down") y un servomotor conectado al pin de salida PWM nº 3 (además de a la alimentación y a tierra), podríamos ejecutar el siguiente código. Gracias a él, pulsando un botón el servomotor se movería en un sentido de giro, y pulsando el otro botón se movería en sentido contrario.

Ejemplo 6.25

```
#include <Servo.h>
Servo miservo;
int pos = 90; //Posición inicial del servo (en el centro)
void setup() {
```

```

pinMode(7, INPUT);
pinMode(8, INPUT);
miservo.attach(3);
miservo.write(pos);

}

void loop() {
    if(digitalRead(7) == HIGH) {
        if( pos > 0) {
            pos--;
            //Mueve el servo de 180 a 0 grados
            miservo.write(pos);
        }
    } else if(digitalRead(8) == HIGH) {
        if( pos < 180) {
            pos= pos++;
            //Mueve el servo de 0 a 180 grados
            miservo.write(pos);
        }
    }
}
}

```

Otro juego sencillo que podemos realizar con solo dos pulsadores (conectados a los pines de entrada digital nº 7 y nº 8, respectivamente, además de a la alimentación y a tierra a través de una resistencia "pull-down") es el de un temporizador, donde un botón servirá para poner en marcha la cuenta de tiempo y el otro para pararla. A través del canal serie se mostrará el número de horas, minutos y segundos transcurridos entre ambas pulsaciones.

Ejemplo 6.26

```

unsigned long inicio, fin, transcurrido;
void setup(){
    Serial.begin(9600);
    pinMode(7, INPUT); //Botón de inicio
    pinMode(8, INPUT); //Botón de fin
}
void loop(){
    if (digitalRead(7)==HIGH){
        inicio=millis();
        delay(200); //Para hacer el "debounce"
    }
    if (digitalRead(8)==HIGH){
        fin=millis();
        delay(200); //Para hacer el "debounce"
    }
}

```

```
    verResultado();
}
}

void verResultado(){
    float h,m,s,ms;
    unsigned long resto;
    transcurrido=fin-inicio;
    h=int(transcurrido/3600000); //Número de horas
    //Obtengo el resto de ms sobrantes que no llegan a una hora
    resto=transcurrido%3600000;
    m=int(resto/60000); //Número de minutos
    //Obtengo el resto de ms que no llegan a un minuto
    resto=resto%60000;
    s=int(resto/1000); //Número de segundos
    //Obtengo el resto de ms que no llegan a un segundo
    ms=resto%1000;
    Serial.print("Total de milisegundos transcurridos");
    Serial.println(transcurrido);
    Serial.print("Tiempo transcurrido formateado");
    Serial.print(h); Serial.print("h ");
    Serial.print(m); Serial.print("m ");
    Serial.print(s); Serial.print("s ");
    Serial.print(ms); Serial.println("ms");
    Serial.println();
}
```

Keypads digitales



Un paso más allá en el uso de pulsadores digitales son los teclados numéricos como el de la imagen del costado (también llamados "keypads"). Estos aparatos no son más que conjuntos de pulsadores conectados entre sí de tal forma que, cuando alguno de ellos es apretado, envía una señal identificativa a nuestra placa Arduino para que esta pueda reaccionar, dependiendo del botón pulsado, de la forma que nosotros deseemos. Una aplicación práctica de estos dispositivos es, por ejemplo, la introducción de una secuencia determinada de dígitos (una "contraseña")

para compararla con una ya previamente definida en el código y así activar (o no) algún componente del circuito.

Si observamos el dorso de un keypad digital típico, veremos que ofrece una serie de pines numerados. Concretamente, si el keypad es de 3x4 botones (como el producto nº **8653** de Sparkfun o el nº **1824** de Adafruit –ambos similares al mostrado en la imagen– o el producto nº **419** de Adafruit –con la particularidad de ser flexible–) ofrecerá siete pines, donde cada uno de ellos se corresponde o bien a una fila de las cuatro existentes, o bien a una columna de las tres. Si el keypad es de 4x4 botones (como el modelo **SPL-012008** de SpikenzieLabs, los ofrecidos por Futurlec en el apartado "Keypads" de su web o el producto **FIT0129** de DFRobot –este último flexible–), ofrecerá 8 pines: 4 para las 4 filas y otros 4 para las 4 columnas. Existen keypads de otras dimensiones, pero no son tan habituales. En cualquier caso, cada uno de los pines que ofrezca el keypad en cuestión se ha de conectar a un pin-hembra digital diferente de la placa Arduino (los cuales actuarán como entradas). Además, dependiendo del modelo de keypad, alguno de sus pines también deberían conectarse al pin "5V" de la placa Arduino para recibir la alimentación (por ejemplo, el producto de Sparkfun necesita ser alimentado a través de sus pines nº 3, 5, 6 y 7 –usando para ello un divisor de tensión de $1K\Omega$ ó $10K\Omega$ – aunque, en cambio, los productos de Adafruit no necesitan ninguna conexión específica a alimentación porque les basta con las siete conexiones a los pines-hembra de Arduino para funcionar).

En todo caso, el sistema de funcionamiento de los keypads siempre es el mismo: cuando se pulsa un botón determinado, el pin correspondiente a su fila y el pin correspondiente a su columna enviarán a la vez una señal digital HIGH a nuestra placa Arduino, identificando inequívocamente a la tecla pulsada, ya que solo existe una pareja posible de valores (fila, columna) para cada botón. Para saber qué pines se corresponden con qué fila o columna, se ha de consultar su datasheet. Si no disponemos del datasheet de nuestro keypad, aún podemos encontrar la relación de los pines con las filas/columnas utilizando la funcionalidad de medir continuidad con un multímetro: se trataría de conectar un cable del multímetro en el pin nº 1 del keypad y otro en el pin nº 2, e ir pulsando los diferentes botones, hasta escuchar (si se produce) el zumbido que indica que hay continuidad: eso nos vinculará el botón pulsado con los dos pines probados en ese momento; entonces pasaríamos a conectar el segundo cable del multímetro al siguiente pin (el nº 3 en este caso) y repetiríamos el proceso. Después conectaríamos ese mismo segundo cable al nº 4, y así hasta el final. Entonces pasaríamos a conectar el primer cable al pin nº 2 y el segundo cable al 3, 4, 5, 6...; luego el primer cable al nº 3 y el segundo cable al 4, 5, 6... y así con todas las parejas de pines posibles, anotando siempre qué botón provoca continuidad en una combinación dada. De esta forma podremos asociar, finalmente, cada pin a una fila o columna concreta. Es posible que en este procedimiento se descubra que hayan pines que no "sirvan para nada"; se pueden obviar tranquilamente.

EL MUNDO GENUINO-ARDUINO

Para facilitar el uso de este tipo de dispositivos, lo más recomendable es utilizar la librería "Keypad" (<https://github.com/Chris--A/Keypad>). Es una librería no bloqueante (es decir: las pulsaciones realizadas en el keypad no interrumpen el funcionamiento normal del microcontrolador) y permite pulsaciones de múltiples teclas a la vez. Una vez instalada como cualquier otra librería, y conectado cada pin del keypad a un pin-hembra digital de la placa Arduino configurado como entrada, ya podemos empezar a usarla. Su documentación oficial (junto con varios ejemplos de código) se encuentra disponible en <http://playground.arduino.cc/Code/Keypad> pero como referencia para el lector a continuación mostramos un sketch de muestra:

Ejemplo 6.27

```
#include <Keypad.h>
#include <Key.h> //Hay que incluir esta línea también
//Suponemos que nuestro keypad es de 3x4
const byte FILAS = 4;
const byte COLUMNAS = 3;
/*Asignamos en un array de arrays cada tecla del keypad a un carácter concreto
que pueda devolver el método keypad.getKey() */
char botones[FILAS][COLUMNAS] = {{'1','2','3'}, {'4','5','6'}, {'7','8','9'}, {'*','0','#'}};
//Definimos qué pin digital Arduino está conectado a cada fila/columna
byte pinsFilas[FILAS] = {5, 4, 3, 2};
byte pinsColumnas[COLUMNAS] = {8, 7, 6};
//Generamos el objeto "keypad" que nos permitirá gestionar el dispositivo en nuestro sketch
Keypad keypad = Keypad( makeKeymap(botones), pinsFilas, pinsColumnas, FILAS, COLUMNAS );
//Establecemos la contraseña válida (de seis cifras)
byte contra[6]={1,2,3,4,5,6};
//Array donde guardaremos la contraseña introducida por el usuario (a comparar con contra[])
byte entradas[6]={0,0,0,0,0,0};
byte numEnt=0;
void setup() { Serial.begin(9600); }
void loop() {
    char tecla = keypad.getKey(); //A cada iteración compruebo si se ha pulsado una tecla nueva
    if (tecla != NO_KEY) { //Si es así (es decir, si se ha obtenido un carácter de los definidos)...
        delay(50); //Para hacer el debounce
        switch(tecla){ //...mira qué carácter concreto es y actúa en consecuencia
            case '*': //Reset": hago limpieza en memoria para poder volver a ...
                limpioContra(); //...empezar la introducción de una nueva contraseña desde cero
                break;
            case '#': //Doy la señal de confirmar la contraseña introducida (y hago limpieza también)
                checkContra();
                limpioContra();
                break;
            default: //Guardo la cifra introducida (así hasta seis)
                if (numEnt <= 5 ) {
```

```

        entradas[numEnt]=tecla;
        numEnt++;
    } else {
        Serial.println("Ha superado el nº de dígitos posibles (6). Pulse '*' para restart");
    }
}
}

void limpioContra(){
    //Borro la contraseña introducida guardada en entradas[]
    for (byte i=0; i<6; i++) { entradas[i]=0; }
    /*Reseteo el índice de entradas[] para volver a empezar la introducción
    de una nueva contraseña desde la primera posición */
    numEnt=0;
}

void checkContra() {
    byte contEntCorrectas=0; /*Variable que cuenta el número de valores iguales -y en la misma
    posición- que hay entre entradas[] y contra[]. Sólo si es 6 se dará por buena la contraseña introducida */
    //Hago la comparación
    for (byte i = 0; i < 6 ; i++ ) {
        if (entradas[i]==contra[i]) { contEntCorrectas++; }
    }
    if (contEntCorrectas==6) {
        Serial.println("CONTRASEÑA CORRECTA");
        //...aquí vendría el código a ejecutar cuando la contraseña es introducida correctamente
    } else {
        Serial.println("CONTRASEÑA INCORRECTA");
        //...aquí vendría el código a ejecutar cuando la contraseña es introducida INCORRECTAMENTE
    }
}
}

```

USO DE LAS ENTRADAS Y SALIDAS ANALÓGICAS

Las funciones que sirven para gestionar entradas y salidas analógicas son las siguientes:

analogWrite(): envía un valor que representa una señal PWM (especificado como segundo parámetro) a un pin digital configurado como OUTPUT (especificado como primer parámetro). Este valor es de tipo *byte*, por lo que solo puede estar dentro del rango de 0 a 255. No todos los pines digitales pueden generar señales PWM: en la placa Arduino UNO por ejemplo solo son los pines 3, 5, 6, 9, 10 y 11 (están marcados en la placa). Cada vez que se ejecute esta función se regenerará la señal. Esta función no tiene valor de retorno.

EL MUNDO GENUINO-ARDUINO

Recordemos que una señal PWM es una señal digital cuadrada que simula ser una señal analógica. El valor simulado de la señal analógica dependerá de la duración que tenga el pulso digital (es decir, el valor HIGH de la señal PWM). Si el segundo parámetro de esta función vale 0, significa que su pulso no dura nada (es decir, no hay señal) y por tanto su valor analógico "simulado" será el mínimo (0V). Si vale 255 (que es el máximo valor posible, principalmente porque cada salida PWM tiene una resolución de 8 bits, y por tanto, solo puede ofrecer hasta $2^8=256$ valores diferentes –de 0 a 255, pues–), significa que su pulso dura todo el periodo de la señal (es decir, es una señal continua) y por tanto su valor analógico "simulado" será el máximo ofrecido por la placa (5V). Cualquier otro valor entre estos dos extremos (0 y 255) implica un pulso de una longitud intermedia (por ejemplo, el valor 128 generará una onda cuadrada cuyo pulso es de la misma longitud que la de su estado bajo) y por tanto, un valor analógico "simulado" intermedio (en el caso que nos ocupa, 2,5V).

Es importante recalcar que esta función no tiene nada que ver con los pines analógicos A0, A1, etc., ya que estos solo funcionan como pines analógicos de entrada (mediante el uso de la función *analogRead()*) pero no de salida. Las salidas analógicas en las placas Arduino que no tengan DAC (es decir, todas menos la Due y la Zero) solo se pueden generar utilizando los pines digitales PWM.

analogRead(): devuelve el valor leído del pin de entrada analógico cuyo número (0, 1, 2...) se ha especificado como parámetro. Este valor se obtiene mapeando proporcionalmente la entrada analógica obtenida (que debe oscilar entre 0 y un voltaje llamado voltaje de referencia, el cual por defecto es 5V) a un valor entero entre 0 y 1023. Esto implica que la resolución de lectura es de 5V/1024, es decir, de 0,049V.

Ya comentamos en el capítulo 2 que es posible aumentar la resolución de lectura (es decir, detectar cambios de voltaje de entrada más pequeños) si se reduce el voltaje de referencia. Esto se hace mediante la función *analogReference()*, explicada en los próximos párrafos. También comentamos en el capítulo 2 que este proceso de mapeado lo realiza un convertidor analógico-digital interno (un "ADC") incorporado a la placa, que tiene 6 canales (por eso hay 6 pines analógicos) y que tiene 10 bits de resolución (por eso los valores finales van de 0 a 1023: 2^{10} valores posibles).

Como los pines analógicos por defecto solamente funcionan como entradas de señales analógicas, no es necesario utilizar previamente la función *pinMode()* con ellos. No obstante, estos pines también incorporan toda la funcionalidad de un pin de entrada/salida digital estándar (incluyendo también las resistencias "pull-up"), por lo

que si se necesita utilizar más pines de entrada/salida digitales de los que la placa Arduino ofrece, y los pines analógicos no están en uso, estos pueden ser utilizados como pines de entrada/salida digitales extra de la forma habitual, simplemente identificándolos con un número correlativo más allá del pin 13, que es el último pin digital. Es decir, el pin "A0" sería el número 14, el "A1" sería el 15, etc. Por ejemplo, si quisieramos que el pin analógico "A3" funcionara como salida digital y además enviara un valor BAJO, escribiríamos primero `pinMode(17,OUTPUT);` y luego `digitalWrite(17,LOW);`

Si un pin analógico no está conectado a nada, el valor devuelto por `analogRead()` fluctuará debido a múltiples factores (como, por ejemplo, los valores que puedan tener las otras entradas analógicas, o lo cerca que esté nuestro cuerpo a la placa, etc.). Esto, que en principio no es deseable, lo podemos utilizar sin embargo para algo útil: para establecer semillas de números aleatorios diferentes (y por tanto, aumentar así la aleatoriedad de las diferentes series de números generados). Esto se haría poniendo como parámetro de `randomSeed()`; el valor obtenido en la lectura de un pin analógico cualquiera que esté libre; es decir, por ejemplo así: `randomSeed(analogRead(0));`

Por otro lado, hay que saber que el convertidor analógico-digital tarda alrededor de 100 microsegundos (0,0001s) en procesar la conversión y obtener el valor digital, por lo que el ritmo máximo de lectura en los pines analógicos es de 10000 veces por segundo. Esto hay que tenerlo en cuenta en nuestros sketches.

Solo si disponemos de las placas Zero o Due, podremos hacer servir otras dos funciones más relacionadas con las entradas y salidas analógicas:

analogWriteResolution(): establece, mediante su único parámetro –de tipo `byte`–, la resolución en bits que tendrá a partir de entonces la función `analogWrite()` a lo largo de nuestro sketch. Por defecto, esta resolución siempre es de 8 bits (es decir, los valores admitidos en el segundo parámetro de `analogWrite()` siempre irán por defecto desde 0 a 255), pero las salidas PWM de las placas Zero y Due tienen la particularidad de permitir una resolución de hasta 10 y 12 bits, respectivamente, por lo que si, por ejemplo, en un sketch ejecutado sobre la placa Zero (o Due) aparece en un determinado momento la línea `analogWriteResolution(10);` a partir de entonces los valores admitidos en el segundo parámetro de `analogWrite()` pasan a estar en el rango de 0 a 1023 (es decir, 2^{10} valores diferentes) y, por tanto, el control de la señal se vuelve mucho más fino y preciso. Esta función no devuelve nada.

EL MUNDO GENUINO-ARDUINO

Por otro lado, tanto la placa Arduino Zero como la Arduino Due disponen de Uno o dos, respectivamente, pines-hembra asociados a sendos conversores digital-analógicos (DAC) que permiten utilizar *analogWrite()* para emitir señales realmente analógicas (es decir, sin usar el "truco" del PWM). El DAC presente en la placa Zero tiene, al igual que ocurría con sus líneas PWM, una resolución de 10 bits (admitiendo, por tanto, 1024 valores diferentes en *analogWrite()* si se especifica previamente *analogWriteResolution(10);*) y los dos DACs presentes en la Due tienen, al igual que ocurría también con sus líneas PWM, una resolución de 12 bits (admitiendo, por tanto, 4096 valores diferentes en *analogWrite()* si se especifica previamente *analogWriteResolution(12);*).

Si en *analogWriteResolution()* se especifica una resolución mayor de la que las salidas analógicas de la placa son capaces de admitir (ya sean PWM o DAC), los bits extra serán descartados. Si, en cambio, se especifica una resolución menor, los bits extra se rellenarán con ceros. Por ejemplo: si escribimos *analogWriteResolution(16);*, solo los primeros 10 –Zero– o 12 –Due– bits de cada valor (empezando por la derecha) serán utilizados por *analogWrite()*, y los últimos 4 no se tendrán en cuenta. Si, en cambio, escribimos *analogWriteResolution(8);*, se añadirán automáticamente 4 bits (iguales a 0) a la izquierda del valor de 8 bits, para que así *analogWrite()* pueda escribir, a través de los dos conversores digital-analógicos, un valor de 12 bits.

***analogReadResolution()*:** establece, mediante su único parámetro –de tipo *byte*–, el tamaño en bits del valor que devolverá la función *analogRead()* a partir de entonces a lo largo de nuestro sketch. Por defecto, este tamaño es de 10 bits (es decir, que *analogRead()* devuelve valores entre 0 y 1023), pero las placas Arduino Zero y Due son capaces de manejar tamaños de hasta 12 bits en los valores devueltos por *analogRead()*, por lo que especificando *analogReadResolution(12);* antes de cualquier lectura, podríamos obtener datos dentro de un rango de entre 0 y 4095. Esta función no devuelve nada.

Si en *analogReadResolution()* se especifica un tamaño mayor del que las entradas analógicas de la placa son capaces de devolver, al valor leído se le añadirán por la izquierda bits extra iguales a 0 hasta llegar al tamaño especificado. Si, en cambio, se especifica una resolución menor, los bits leídos sobrantes (por la izquierda) no se tendrán en cuenta y serán descartados.

Ejemplos con salidas analógicas

Hasta ahora hemos conectado siempre los LEDs a salidas digitales. Por tanto, estos solo podían estar en dos estados: o apagados o encendidos. Pero los LEDs

(como tantos otros) son dispositivos analógicos. Esto implica que pueden tener muchos más estados (de hecho, pueden tener estados continuos). Es decir, que pueden iluminarse con muchas intensidades diferentes y de forma gradual.

Para conseguir cambiar la intensidad lumínica de un LED, primero deberemos montar el circuito. La conexión del LED no tiene ningún misterio: su terminal positivo ha de ir enchufado a un pin PWM de nuestra placa Arduino UNO (por ejemplo el nº 9) y su terminal negativo a tierra; uno de sus terminales (es indiferente cuál) además deberá estar conectado en serie a un divisor de tensión (de unos 220 ohmios está bien). A continuación, debemos escribir el sketch; el siguiente, por ejemplo, modifica cíclicamente el voltaje PWM ofrecido por la salida analógica donde esté conectado el LED desde su valor mínimo (0) hasta su valor máximo (255) y de nuevo hasta su valor mínimo (0) para volver a empezar otra vez. El efecto visible será primero una progresiva iluminación del LED para pasar a un apagado progresivo y a continuación empezar otra vez el encendido gradual:

Ejemplo 6.28

```
int brillo = 0;
void setup(){
    pinMode(9, OUTPUT); //El LED está conectado al pin 9 (PWM)
}
void loop(){
    //Incrementa el brillo (de mínimo a máximo)
    for(brillo = 0 ; brillo<= 255; brillo=brillo+5) {
        analogWrite(9, brillo);
        /*Espero 30 milisegundos con la señal actual de analogWrite(). Si se desea que el
         incremento (o disminución) del brillo se realice a otra velocidad, basta con modificar
         este tiempo de espera */
        delay(30);
        //Aquí se escribirán las otras acciones que queramos ejecutar a cada paso del bucle
    }
    //Disminuye el brillo (de máximo a mínimo)
    for(brillo = 255; brillo>=0; brillo=brillo-5) {
        analogWrite(9, brillo);
        delay(30);
        //Otras acciones...
    }
}
```

Otro código que hace exactamente lo mismo es:

EL MUNDO GENUINO-ARDUINO

Ejemplo 6.29

```
int brillo = 0;
int incremento = 5; //Puede valer negativo (decremento)
void setup(){
    pinMode(9, OUTPUT); //El LED está conectado al pin 9 (PWM)
}
void loop() {
    analogWrite(9, brillo);
    delay(30);
    //Otras acciones...
    //Modifico el brillo un valor dado (el resultado se verá al ejecutar el próximo analogWrite())
    brillo = brillo + incremento;
    //Si se llega a los extremos, se invierte la dirección del incremento
    if (brillo == 0 || brillo == 255) {
        incremento = -incremento;
    }
}
```

Si dibujáramos una gráfica donde el eje horizontal representa el tiempo y el vertical el brillo del LED obtenido a partir de ejecutar los ejemplos anteriores 6.28 y 6.29, veríamos que se obtiene una señal triangular (algo serrada porque los incrementos del brillo en realidad no son continuos, pero esto no lo tendremos en cuenta); es decir, el flanco de subida apagado-encendido aparecería como una línea recta ascendente y el flanco de bajada encendido-apagado como otra línea recta, descendente. Esto significa que la velocidad a la que varía el brillo del LED es constante.

Control interactivo (mediante pulsadores)

A continuación presentamos un código donde interviene, además de un LED conectado al pin PWM nº 9, un pulsador en configuración "pull-down" cuyo cable de control está conectado al pin digital nº 8. La novedad está en el bucle *while*: lo que conseguimos con él es que mientras tengamos presionado el pulsador y no hayamos llegado al máximo de brillo, el LED se irá iluminando al ritmo que marque *delay()* debido al incremento del brillo de 5 puntos en cada vuelta. Una vez dejemos de pulsar el botón, de forma brusca apagaremos el LED.

Ejemplo 6.30

```
int brillo=0;
void setup() {
    pinMode(9, OUTPUT);
    pinMode(8, INPUT);
```

```

    }
void loop() {
    while (digitalRead(8) == HIGH && brillo<=255){
        analogWrite(9,brillo);
        delay(200);
        brillo=brillo+5;
    }
    brillo=0;
    analogWrite(9, 0);
}

```

¿Qué pasaría si en el código anterior sustituimos la palabra "while" por la palabra "if"? Que el LED nunca se encenderá, porque aunque la condición se cumpla y se haga el primer *analogWrite()* del interior del "if", justo después de este se vuelve a resetear la variable "brillo" a cero, y por si fuera poco, el último *analogWrite()* apaga lo poco que se había podido iluminar, por lo que en cada repetición del "loop" volveremos a estar en las mismas.

Control interactivo (a través del canal serie)

Podríamos incluso manipular el brillo de un LED a voluntad mediante el envío de algún comando adecuado a través del canal serie. En el siguiente ejemplo supondremos que tenemos 3 LEDs diferentes (de colores primarios rojo, verde y azul, respectivamente) conectados cada uno a través de su divisor de tensión correspondiente a un pin PWM diferente. La idea es utilizar el "Serial monitor" para enviar a la placa Arduino un comando determinado que especifique qué LED en concreto queremos manipular (indicado por la letra "r", "g" o "b") y qué cantidad de brillo le queremos asignar (indicado por un valor entre 0 y 255). Así, si queremos por ejemplo apagar el LED rojo el comando a enviar debería ser "r0". Y si queremos iluminar al máximo el LED azul, el comando debería ser "b255".

Ejemplo 6.31

```

char color; //Es importante que sea de tipo char y no byte
byte brillo;
byte LedRojo = 5;
byte LedVerde = 6;
byte LedAzul = 9;
void setup() {
    Serial.begin(9600);
    pinMode(LedRojo, OUTPUT);
    pinMode(LedVerde, OUTPUT);
    pinMode(LedAzul, OUTPUT);
}

```

EL MUNDO GENUINO-ARDUINO

```
analogWrite(LedRojo, 127); //Establezco un brillo inicial medio
analogWrite(LedVerde, 127);
analogWrite(LedAzul, 127);
}
void loop () {
    //Leo el primer carácter recibido por el canal serie
    if (Serial.available()>0) {
        color=Serial.read();
        if( color == 'r' || color == 'g' || color == 'b' ) {
            //Extraigo el número que sigue a la primera letra
            brillo=byte(Serial.parseInt());
            if(color == 'r') {
                analogWrite(LedRojo, brillo);
            } else if(color == 'g'){
                analogWrite(LedVerde, brillo);
            }else if(color == 'b'){
                analogWrite(LedAzul, brillo);
            }
        }
    }
}
```

Otro código que persigue el mismo objetivo que el anterior, pero de una forma algo más compacta, es el siguiente. En él también enviamos a través del canal serie la cantidad de brillo que deseamos que tenga cada LED, pero lo hacemos enviando siempre el brillo de los tres LEDS en cada "comando". La idea es enviar los tres brillos separados por comas (o cualquier otro carácter que hayamos elegido como "frontera" entre valor y valor), de tal forma que Arduino recoja uno tras otro el primer brillo, el segundo y el tercero, para seguidamente asignarlos a los LEDs adecuados:

Ejemplo 6.32

```
byte brilloRojo, brilloVerde, brilloAzul;
byte LedRojo = 5;
byte LedVerde = 6;
byte LedAzul = 9;
void setup() {
    Serial.begin(9600);
    pinMode(LedRojo, OUTPUT);
    pinMode(LedVerde, OUTPUT);
    pinMode(LedAzul, OUTPUT);
    analogWrite(LedRojo, 127);
    analogWrite(LedVerde, 127);
```

```

        analogWrite(LedAzul, 127);
    }
    void loop () {
        //Leo todos los caracteres detectados en este momento por el canal serie
        while (Serial.available()>0) {
            //Busco el siguiente número entero válido en los datos entrantes
            brilloRojo = Serial.parseInt();
            //Repite la búsqueda
            brilloVerde = Serial.parseInt();
            //Repite la búsqueda
            brilloAzul = Serial.parseInt();
            //Miro si ha llegado el carácter de nueva línea: eso marca el fin del "comando"
            if (Serial.read() == '\n') {
                //Restringo los valores leídos al rango 0-255, por si acaso
                brilloRojo = constrain(brilloRojo, 0, 255);
                brilloVerde = constrain(brilloVerde, 0, 255);
                brilloAzul = constrain(brilloAzul, 0, 255);
                //Ilumino los LEDs convenientemente
                analogWrite(LedRojo, brilloRojo);
                analogWrite(LedVerde, brilloVerde);
                analogWrite(LedAzul, brilloAzul);
            }
        }
    }
}

```

Podemos jugar a establecer los valores PWM de los tres LEDs anteriores (rojo, verde y azul) de forma que se puedan obtener otros colores. Para ello es necesario colocar los tres LEDs físicamente muy próximos y es recomendable, para conseguir un mejor efecto, difuminar la luz a través de algún material tal como pañuelos de papel, por ejemplo. Si variamos entonces el brillo de uno de los LEDs (o de dos o de los tres a la vez) alteraremos la combinación total y obtendremos, a partir de los tres colores primarios, una mezcla que producirá un color totalmente diferente (amarillo, lila, naranja...). Incluso se pueden generar transiciones de un color a otro si vamos modificando el brillo de cada LED (mediante bucles *for*, por ejemplo).

Uso de LEDs RGB

Para generar colores no primarios, también se puede utilizar un LED RGB, que no es más que un LED con cuatro terminales. Cuidado porque pueden ser de dos tipos: si es de tipo "cátodo común" (como el producto nº **9264** de Sparkfun –de tipo "difuso"– o los productos nº **105** de Sparkfun o **FIT0095** de DFRobot –de tipo "claro"–), el terminal más largo se conecta a tierra, y los tres restantes se conectan (a través de sendos divisores de tensión) a diferentes pines PWM de nuestra placa

EL MUNDO GENUINO-ARDUINO

Arduino: uno servirá para recibir (usando un divisor de 220Ω por ejemplo) la intensidad deseada de color rojo, otro para recibir (usando otro divisor de 100Ω por ejemplo) la intensidad de color verde y otro para recibir (usando otro divisor de 100Ω) la intensidad de color azul. Si es de tipo "ánodo común" (como el producto nº **159** de Adafruit –de tipo "difuso"– o el producto nº **10820** de Sparkfun –de tipo "claro"–), el terminal más largo se conecta a la alimentación (5V) y los tres restantes se conectan (igualmente a través de un divisor de tensión en serie) a tierra y además a diferentes pines PWM de nuestra placa Arduino. Hay que tener en cuenta además que en los LEDs de ánodo común, el valor enviado mediante *analogWrite()* está invertido: un valor 0 hace brillar al máximo el color correspondiente y un valor 255 lo apaga. Un código que muestra el uso de un LED RGB es el siguiente:

Ejemplo 6.33

```
int pinRojo = 6;
int pinVerde = 5;
int pinAzul = 3;
void setup () {
    pinMode(pinRojo, OUTPUT);
    pinMode(pinVerde, OUTPUT);
    pinMode(pinAzul, OUTPUT);
}
void loop () {
    setColor(0, 0, 0);          // Apagado
    setColor(255, 0, 0);        // Rojo
    setColor(0, 255, 0);        // Verde
    setColor(0, 0, 255);        // Azul
    setColor(0, 255, 255);      // Cian
    setColor(255, 255, 0);      // Amarillo
    setColor(255, 0, 255);      // Fucsia
    setColor(255, 255, 255);    // Blanco
}
void setColor (int rojo, int verde, int azul) {
    //Cátodo común
    //analogWrite(pinRojo, rojo);
    //analogWrite(pinVerde, verde);
    //analogWrite(pinAzul, azul);
    //Ánodo común
    analogWrite(pinRojo, 255 - rojo);
    analogWrite(pinVerde, 255 - verde);
    analogWrite(pinAzul, 255 - azul);
    delay(1000);
}
```

Una utilidad práctica de los LEDs RGB es la de mostrar información (generalmente obtenida del entorno en tiempo real) de una forma cualitativa e intuitiva. Por ejemplo, según la temperatura detectada por el sensor pertinente (a estudiar en el próximo capítulo) un LED RGB podría iluminarse de color rojo (si hace mucho calor) o de color azul (si hace mucho frío) o de otros colores (si hace una temperatura intermedia).

Otra idea menos útil pero muy llamativa es la de enviar valores PWM aleatorios a uno o varios LEDs independientes (usando *random()*; como segundo parámetro de *analogWrite()*;) y entre envío y envío esperar también un tiempo aleatorio (usando *random()*; como parámetro de *delay()*). Realmente el efecto es psicodélico.

Ejemplos con entradas analógicas (potenciómetros)

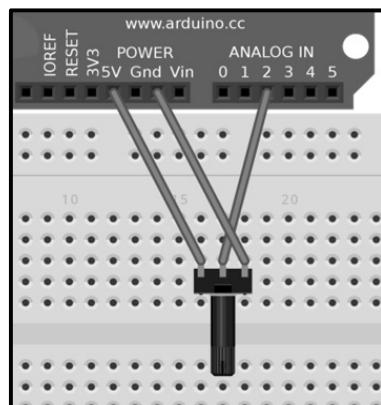
Los ejemplos del apartado anterior están muy bien, pero nos gustaría modificar el brillo del LED a voluntad. Para ello, en nuestros circuitos podríamos utilizar un potenciómetro conectado por un extremo a una fuente de voltaje conocido (los 5V ofrecidos por la propia placa Arduino ya nos va bien), por otro a tierra y por su patilla central a algún pin de entrada analógico de la placa. La idea es que la placa reciba (proveniente de esa patilla central) una señal analógica controlable a voluntad, y aprovechar esto para "reenviarla" a las salidas analógicas de la placa que queramos. En otras palabras: usaremos un potenciómetro conectado a una entrada analógica como intermediario para manipular un dispositivo conectado a alguna salida PWM (como un LED). Si no decimos lo contrario, en nuestros proyectos utilizaremos un potenciómetro de 10KΩ.

Ya sabíamos que un potenciómetro es una resistencia variable controlable por su patilla central, pero ¿cuál es esta señal analógica que recibe la placa Arduino proveniente de él? Esta señal es el voltaje existente entre la patilla central y el extremo del potenciómetro conectado a tierra. Por pura y simple Ley de Ohm, al variar la resistencia existente entre la patilla central y sus extremos, varía también la caída de potencial entre estos puntos. Concretamente, cuando entre la patilla central y el extremo del potenciómetro conectado a la alimentación haya una resistencia cercana a cero (y por tanto la resistencia entre la patilla central y el otro extremo, el conectado a tierra, sea máxima), el voltaje leído por la entrada analógica será cercano a 5V. Cuando la patilla central esté en el otro lado, tocando al extremo conectado a tierra, la lectura será cercana a 0V. Este valor controlable del voltaje (analógico) leído es lo que hemos dicho que podremos "reenviar" a través de los pines PWM de la placa a los dispositivos que deseemos controlar analógicamente.

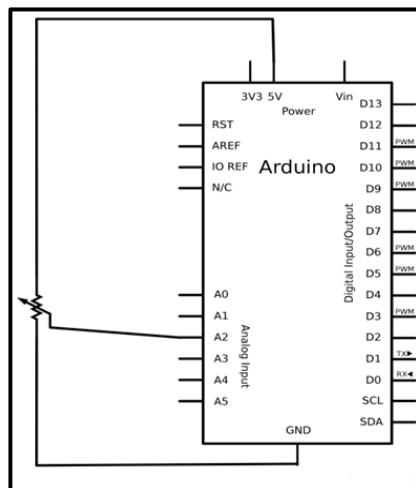
EL MUNDO GENUINO-ARDUINO

Recordemos por otro lado que la placa Arduino dispone de un conversor analógico-digital que solo permite utilizar valores entre 0 y 1023, por lo que el valor máximo del voltaje leído a través del potenciómetro (es decir, los 5V) es convertido siempre al valor numérico 1023 (y el mínimo, lógicamente a 0). Los valores intermedios son convertidos proporcionalmente según la cantidad de voltaje recibido por el pin.

Empezaremos por un circuito muy sencillo para ver en la práctica todo lo que se acaba de explicar. Consta tan solo de un potenciómetro y nada más, conectado tal como se ha dicho. El pin de entrada analógico elegido ha sido el nº 2.



El esquema eléctrico es realmente simple:



Si ejecutamos el siguiente sketch y movemos el potenciómetro, veremos por el "Serial monitor" las lecturas realizadas a través del pin analógico 2, que no son más que valores entre 0 y 1023. Notar que no se ha utilizado la función `pinMode()` como hasta ahora porque los pines-hembra analógicos de la placa Arduino solo pueden ser de entrada.

Ejemplo 6.34

```
int valorPot=0;
void setup(){
    Serial.begin(9600);
}
void loop(){
    valorPot=analogRead(2);
    Serial.println(valorPot);
    delay(100);
}
```

De todas formas, tendría más gracia observar el valor analógico correspondiente a esa lectura. Es decir, ya sabemos que si vemos un 1023 este valor se corresponde con 5V (esto es solo porque suponemos que estamos alimentando el potenciómetro con 5V –los ofrecidos por la propia placa–), pero ¿y si vemos un 584? ¿Cuántos voltios se reciben en ese caso por la entrada analógica? Para saberlo, simplemente debemos aplicar una regla de proporcionalidad: multiplicar el valor leído por 5/1023.

Ejemplo 6.35

```
int valorPot=0;
float voltajePot=0;
void setup(){ Serial.begin(9600); }
void loop(){
    valorPot=analogRead(2);
/*La siguiente línea convierte el valor de valorPot en un valor de voltaje. Fijarse que los valores obtenidos de analogRead() van desde 0 a 1023 y los valores que queremos van desde 0 a 5. Podríamos pensar en utilizar la función map(), pero esta solo devuelve valores enteros, por lo que no nos sirve. Afortunadamente, como en este caso particular los mínimos de ambos rangos son 0, en vez de map() podemos escribir una simple regla de proporcionalidad. Hay que tener en cuenta el detalle de haber especificado los valores numéricos de la fórmula como de tipo "float" (añadiéndoles el 0 decimal) para que el resultado obtenido sea de tipo "float" también y no se trunque.*/
    voltajePot=valorPot*(5.0/1023.0);
    Serial.println(voltajePot);
    delay(100);
}
```

EL MUNDO GENUINO-ARDUINO

Con un poco de imaginación, podríamos modificar el ejemplo anterior para que en vez de mostrar el voltaje leído por el canal serie, lo visualizáramos en una tira de LEDs (10 por ejemplo), a modo de "termómetro" luminoso. Se deja como ejercicio.

Medias y calibraciones

Cuando trabajamos con sensores analógicos uno de los problemas que podemos tener es que cada cierto tiempo obtengamos algún "pico", es decir, un valor distorsionado y separado de los demás, y por tanto, erróneo. Es decir, ruido. Para intentar "suavizar" los valores leídos por si aparece alguno demasiado errático, podemos utilizar un truco: leer la entrada analógica repetidas veces y calcular la media de los valores leídos, considerando esta el valor medido fiable.

Para probar esta manera de obtener datos podemos utilizar el mismo circuito del ejemplo anterior. Concretamente, el código siguiente guarda diez lecturas analógicas en un array de diez posiciones, una a una. Por cada nuevo valor guardado, suma todos los valores y divide el resultado por el número de elementos del array (es decir, calcula la media de esos valores en ese preciso momento). Esta media, mostrada en el "Serial monitor", ofrece una lectura más suavizada del conjunto de valores leídos. Como la media se calcula cada vez que se lee un nuevo valor (en vez de esperar a llenar el array de diez valores nuevos, que sería otra manera), no se aprecia ningún tiempo de espera en los cálculos. Lógicamente, cuanto mayor sea el número de elementos del array, mayor suavizado habrá en el resultado final, pero también será más lenta la obtención de este.

Ejemplo 6.36

```
const int numElementos = 10; //Número de elementos del array
int lecturas[numElementos]; //Aquí se guardan las lecturas
int index = 0; //Índice para irse moviendo por los elementos
//Valor de la suma de los 10 valores que haya en un momento dado en el array
int total = 0;
int media = 0; //Es igual a total/numElementos
void setup(){
    Serial.begin(9600);
    //Inicializo todos los elementos del array a 0
    for (i = 0; i < numElementos; i++){ lecturas[i] = 0; }
}
void loop() {
/*Quito de la suma total el valor que será sobrescrito enseguida por el nuevo valor obtenido. De esta forma, se mantiene tan solo la suma de los valores que en este momento estén dentro del array */
    total= total - lecturas[index];
    lecturas[index] = analogRead(2);
```

```

//Añado el valor recién leído a la suma total
total= total + lecturas[index];
/*Avanzo a la siguiente posición del array para
sobrescribir en esta nueva posición el próximo valor leído*/
index = index + 1;
//Si estamos en el final del array...
if (index >= numElementos){
    //...vuelvo al principio para sobrescribir por allí
    index = 0;
}
//Calculo la media de los 10 valores actuales
media = total / numElementos;
Serial.println(media);
delay(1); //Me espero para leer de nuevo (por estabilidad)
}

```

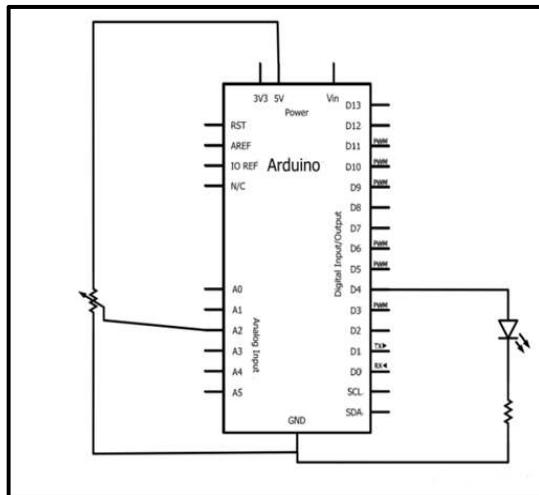
Entradas y salidas

Volvamos al ejemplo 6.35: ya sabemos obtener un dato (de hasta 1024 valores posibles diferentes) proporcional al voltaje analógico recibido. La gracia está ahora en utilizar este valor de tensión para aplicársela a los dispositivos conectados a las salidas PWM de la placa Arduino. De esta manera, podremos variar gradualmente el brillo de un LED, la velocidad de un motor, la frecuencia de un sonido emitido por un zumbador, etc., según les enviamos más o menos voltaje.

Por suerte, tanto el rango de voltaje analógico leído de entrada es de 5V (porque hemos alimentado el potenciómetro con la propia placa) como también lo es el rango del voltaje de salida ofrecidos por los pines-hembra de Arduino, así que en este sentido no hay que preocuparse de que los valores leídos por un lado no sean eléctricamente seguros para ser utilizados en el otro. No obstante, hay que tener en cuenta que los valores leídos pueden estar entre 0 y 1024 (como ya hemos dicho) pero los valores PWM recordemos que solo pueden estar entre 0 y 255. Esto es muy importante, porque nos obliga en nuestro sketch a hacer un "mapeo" (normalmente con *map()*) de los valores obtenidos para que se ajusten a este nuevo rango cuatro veces más pequeño que el original.

Todo esto lo probaremos en el siguiente circuito, donde añadiremos un LED (conectado a un pin de la placa que ha de ser de tipo PWM, como por ejemplo el nº 9) al potenciómetro que ya teníamos montado:

EL MUNDO GENUINO-ARDUINO



Y aquí está el código:

Ejemplo 6.37

```
int valorPot=0;
void setup(){
    pinMode(9,OUTPUT);
}
void loop(){
    valorPot=analogRead(2);
/*Los valores de analogRead() van desde 0 a 1023 y los valores de digitalWrite van desde 0 a 255, por
eso reajustamos el valor leído para poderlo reenviar. En este caso particular, como los mínimos de
ambos rangos son 0, en vez de map() podríamos haber escrito una simple regla de proporcionalidad,
mediante la fórmula valorPot=valorPot*(255.0/1023.0), o dicho de otra forma: dividiendo el valor
original entre 4.0 (el 4 ha de ser un valor "float" para que el resultado no se trunque!, de ahí el 0
decimal).*/
    valorPot=map(valorPot,0,1023,0,255);
    digitalWrite(9,valorPot);
    //Espero un rato para que la señal de digitalWrite se mantenga
    delay(100);
}
```

En vez de utilizar la lectura del potenciómetro para variar de forma continua el brillo de un LED, otra cosa que podemos hacer con el mismo circuito del ejemplo anterior es enviar al LED una señal digital con *digitalWrite()* –es decir, sin valores intermedios: o se enciende (HIGH) o se apaga (LOW)– para hacerlo parpadear y utilizar entonces la lectura del potenciómetro como parámetro de *delay()* para

establecer el tiempo de parpadeo. De esta forma, al variar de forma continua el estado del potenciómetro, variaremos de forma continua el tiempo de parpadeo:

Ejemplo 6.38

```
int valorPot = 0;
void setup() {
    pinMode(9, OUTPUT);
}
void loop() {
    valorPot = analogRead(2);
    digitalWrite(9, HIGH);
    delay(valorPot);
    digitalWrite(9, LOW);
    delay(valorPot);
}
```

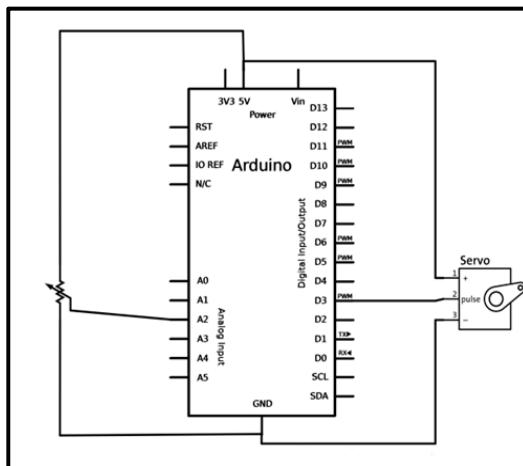
O también encender el LED solamente si el valor leído del potenciómetro supera un determinado umbral:

Ejemplo 6.39

```
int valorPot = 0;
void setup() {
    pinMode(9, OUTPUT);
}
void loop() {
    valorPot = analogRead(2);
    if (valorPot > 500) {
        digitalWrite(9, HIGH);
    } else {
        digitalWrite(9, LOW);
    }
    delay(valorPot);
}
```

Evidentemente, además de LEDs, mediante un potenciómetro podemos controlar cualquier otro tipo de actuador, como por ejemplo un servomotor. Si diseñamos un circuito tal como el mostrado en la figura siguiente, podríamos girar el servomotor un ángulo determinado, según el valor leído del potenciómetro.

EL MUNDO GENUINO-ARDUINO



El código necesario para ello sería el siguiente:

Ejemplo 6.40

```
#include <Servo.h>
Servo miservo;
int valorPot = 0;
void setup() {
    miservo.attach(3);
}
void loop() {
    valorPot = analogRead(0);
/*Los valores de analogRead() van desde 0 a 1023 y los valores aceptados por miservo.write() van
desde 0 a 180, por eso reajustamos el valor leído para poderlo utilizar con el servomotor. En este caso
particular, como los mínimos de ambos rangos son 0, en vez de map() podríamos haber escrito una
simple regla de proporcionalidad, mediante la fórmula valorPot=valorPot*(180.0/1023.0)*/
    valorPot = map(valorPot, 0, 1023, 0, 180);
    miservo.write(valorPot);
    delay(15); //Para dar tiempo al servo a moverse
}
```

Ejemplo de uso de joysticks como entradas analógicas

Un joystick internamente no es más que un conjunto de dos potenciómetros que permiten medir el movimiento de la palanca a lo largo del eje X y el eje Y (es decir, en 2 dimensiones). Por tanto, las conexiones necesarias para utilizar un joystick con nuestra placa Arduino son las siguientes: un extremo de cada potenciómetro ha de estar conectado a la fuente de alimentación del circuito (normalmente, los 5V

proporcionados por la placa Arduino), el otro extremo de cada potenciómetro ha de estar conectado a tierra, y la patilla central de cada potenciómetro ha de estar conectado a una entrada analógica diferente.

Para saber el desplazamiento realizado en un eje o en otro, deberemos consultar el valor leído en la entrada analógica correspondiente (el cual puede ir desde 0 en un extremo hasta 1023 en otro). Dependiendo de la magnitud de esos dos valores leídos, mediante "ifs" o "switches" podremos decidir entonces qué hacer.

La mayoría de veces, el joystick también incorpora un pulsador interno, que se activa al apretarlo. En estos casos, para detectar estas pulsaciones generalmente deberemos conectar además otro cable a nuestra placa Arduino, pero esta vez a una entrada digital.

IteadStudio distribuye un módulo, el "**Joystick breakout module**", cuya conexión es muy sencilla. Tan solo ofrece 5 pines: "+" (a conectar a 5V), "G" (a conectar a tierra), "X" (a conectar a una entrada analógica), "Y" (a conectar a la otra entrada analógica) y "B" (a conectar a una entrada digital, para detectar pulsaciones). Sus dos potenciómetros internos son de 10KΩ cada uno. Otros productos parecidos son el **nº 512** de Adafruit o el **nº 9032** de Sparkfun. Si, no obstante, queremos un joystick similar a los anteriores pero sin pulsador integrado, podemos adquirir el producto **nº 245** de Adafruit o el **nº 27800** de Parallax (con este último las conexiones a realizar son: pines "L/R+" y "V/D+" a alimentación de 5V, pin "L/R" a una entrada analógica, "V/D" a la otra entrada analógica y "GND" a tierra).

También podemos utilizar shields que incluyen un joystick y varios pulsadores extra más. Se debe consultar la documentación de cada shield para saber a qué entradas analógicas y digitales está vinculado tanto el joystick como los diversos pulsadores incluidos, de manera que podamos escribir nuestros sketches correctamente. Ejemplos de este tipo de shield son: el "**InputShield**" de LiquidWare, el "**Arduino Input Shield**" de DFRobot, el "**Joystick Shield**" de IteadStudio, el "**Joystick Shield**" de Elecfreaks o el "**Joystick Shield Kit**" de Sparkfun (aunque este último, tal como su nombre indica, viene en forma de kit, por lo que es necesario soldar sus partes).

A continuación, se muestra un código muy sencillo válido para todos los módulos y shields anteriores, donde, mientras se manipula un joystick, se va visualizando en tiempo real por el "Serial monitor" los valores leídos por las entradas analógicas nº 0 (eje X) y nº 1 (eje Y).

EL MUNDO GENUINO-ARDUINO

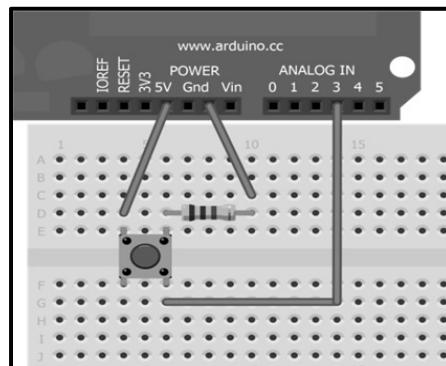
Ejemplo 6.41

```
byte ejeX=0;           //Entrada analógica eje X (nº 0)
byte ejeY=1;           //Entrada analógica eje Y (nº 1)
byte boton=3;          //Entrada digital botón (nº 3)
int valorx,valorY,valorboton; //Valores leídos
void setup() {
    Serial.begin(9600);
}
void loop(){
    valorx = analogRead(ejeX);
    Serial.print( "X:" );
    Serial.print(valorx);

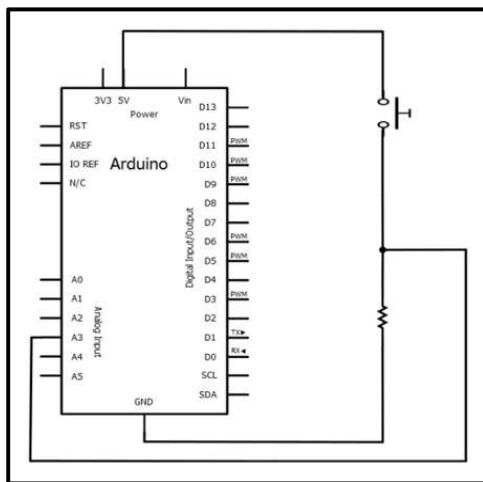
/*Es necesario hacer una pequeña pausa entre lecturas de diferentes pines analógicas porque si no
puede ser que obtengamos la misma lectura dos veces (debido al funcionamiento interno de esos pines)
*/
    delay(100)
    valorY = analogRead(ejeY);
    Serial.print ( " | Y:" );
    Serial.print (valorY);
    delay(100);
    valorboton = digitalRead(boton);
    Serial.print ( " | Botón: " );
    Serial.print (valorboton);
}
```

Ejemplo de uso de pulsadores como entradas analógicas

Los pulsadores se pueden utilizar para detectar valores diferentes más allá de los simples HIGH y LOW, pero para ello deberemos conectarlos a entradas analógicas. Montaremos el siguiente circuito (usando la configuración con resistencia "pull-down"):



Como se puede ver, es muy parecido al que ya vimos cuando tratamos las entradas digitales, solo que ahora el "cable de control" del pulsador está conectado a una entrada analógica de la placa (concretamente, la número 3) en vez de a un pin digital. El esquema eléctrico correspondiente sería:



Para probar el circuito anterior, podemos utilizar un sketch muy parecido al que vimos cuando introdujimos por primera vez las entradas digitales, pero ahora estaremos viendo lo que ocurre en una entrada analógica.

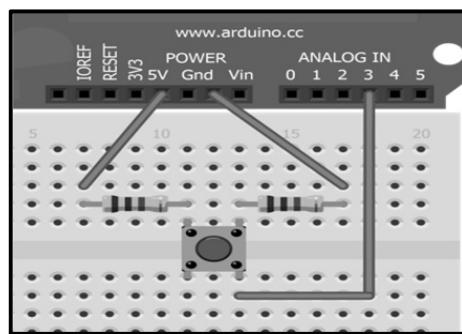
Ejemplo 6.42

```
int estadoBoton=0; //Guardará el estado "analógico" del botón
void setup(){ Serial.begin(9600); }
void loop() {
    estadoBoton=analogRead(3);
    Serial.println(estadoBoton);
    delay (50); //Para mayor estabilidad entre lecturas
}
```

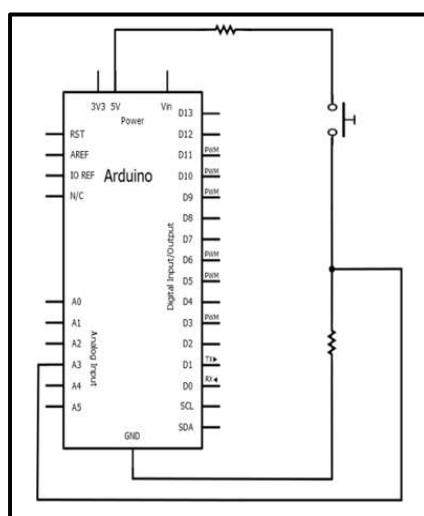
¿Y qué es lo que se ve? Que en vez de obtener un valor HIGH (1) cuando se pulsa el botón y un valor LOW (0) cuando no, en el primer caso se obtiene un valor 1023 (es decir, el equivalente, tras la conversión analógico-digital interna) a 5V, y en el segundo un valor 0 (0V). ¿Por qué? Porque cuando se pulsa el botón el voltaje existente entre la fuente de alimentación (los 5V proporcionados por la placa) y el pin de entrada analógico es precisamente 5V, ya que hay un camino directo entre ambos extremos sin ninguna caída de potencial entremedio. Pero cuando el botón está abierto, el pin analógico está conectado a tierra a través de la resistencia "pull-down".

EL MUNDO GENUINO-ARDUINO

Si ahora queremos leer un valor de voltaje entre 0 y 5V (es la gracia de tener lecturas analógicas!), deberíamos introducir un divisor de tensión en el camino entre la fuente de alimentación y el pin analógico de entrada. Concretamente, colocaremos una resistencia entre el pin de 5V de la placa y la patilla del pulsador conectado a ella, tal como se muestra en las siguientes figuras. El valor de la resistencia puede ser cualquiera, pero cuanto mayor sea, mayor caída de potencial provocará entre sus bornes, y por tanto, menor será la lectura que reciba el pin analógico cuando se pulse el botón. Así pues, tenemos:



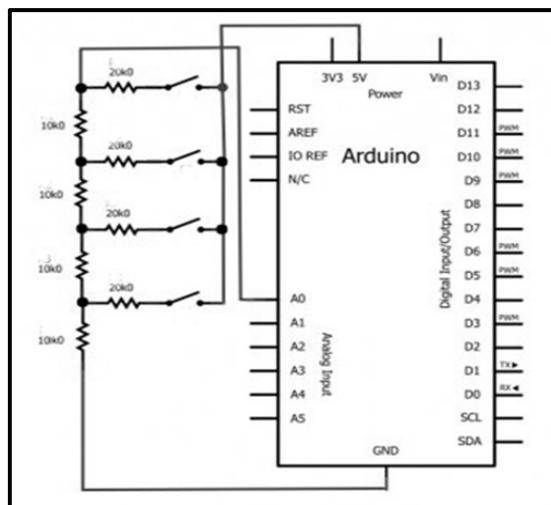
Es decir:



En el caso de haber utilizado la configuración del pulsador con resistencias "pull-up", entonces deberíamos haber colocado el divisor de tensión entre tierra y la patilla del pulsador conectada a esta.

¿Qué ocurre ahora si volvemos a ejecutar el mismo sketch de antes? Que ahora veremos un valor intermedio entre 0 y 1023, dependiendo del valor de la resistencia que hayamos puesto como divisor de tensión (recordemos, a mayor resistencia, el valor leído será menor). Ahora mismo, por tanto, tenemos un circuito que recibe una señal analógica (de un valor fijo) solo cuando se mantiene pulsado el botón.

¿Para qué nos puede servir esta funcionalidad de los pulsadores? Para por ejemplo poder implementar un circuito R-2R "en escalera", tal como el mostrado en la página siguiente. Este tipo de circuito utiliza varios pulsadores, todos ellos conectados a un único pin de entrada analógica. Cada uno de estos pulsadores cuando es presionado enviará una señal analógica de un determinado voltaje a la entrada a la que están conectados (en este caso, la nº 0). La gracia está en que dependiendo del pulsador que presionemos, el valor del voltaje de la señal recibida por el pin analógico será diferente. Incluso se pueden pulsar varios botones a la vez, y esto generará otro voltaje diferente (aunque es posible que algún conjunto concreto de pulsaciones se solape con otro). Todo esto es gracias al diseño R-2R "en escalera" de resistencias "pull-down" y divisores de tensión, tal como el mostrado en la figura siguiente. Este diseño se llama así porque las resistencias "pull-down" (mostradas en la figura verticalmente) tienen siempre la mitad de valor (en este ejemplo, 10KΩ) que los divisores de tensión (mostrados en la figura horizontalmente, y, en este ejemplo, con un valor de 20KΩ). Es preferible que todas las resistencias tengan un 1% de tolerancia para mejorar la exactitud de las medidas.



EL MUNDO GENUINO-ARDUINO

Una vez realizado el circuito R-2R, lo único que tendríamos que hacer para aprovechar estas diversas entradas es escribir un sketch que fuera comprobando (con varios "ifs" o un "switch") qué valor de voltaje recibe la placa, para así responder según lo que proceda. De esta manera, podríamos implementar un "keypad" artesano con solo un pin de nuestra placa.

De hecho, una plaquita que implementa esta misma idea es la llamada "**ADKeyboard Module**" de DFRobot. Esta plaquita contiene 5 botones que tan solo requieren el uso de una entrada analógica de nuestra placa Arduino para ser monitorizados de forma independiente. En la página web de este producto se ofrece además un código Arduino de ejemplo para saber cómo se puede realizar la gestión de las pulsaciones.

Otra plaquita similar es la "**TWI Keyboard**" de Akafugu. No obstante, esta plaquita se comunica con nuestra placa Arduino vía I²C y por tanto, requiere 2 cables (además del de alimentación y tierra). También dispone de 5 botones independientes, los cuales pueden ser controlados mediante la librería descargable de <https://github.com/akafugu/twikeyboard>. Esta librería es capaz de detectar los eventos de pulsación, soltado y repetición de los botones, y gestiona correctamente el efecto de "bounce".

También podemos utilizar el producto nº **27801** de Parallax, el cual es un pulsador de cinco posiciones. Es decir, funciona como si fuera una plaquita de cinco pulsadores, aunque a simple vista se asemeja a un joystick. Su conexión es sencilla: el pin marcado como "VCC" va a 5V, el "GND" va a tierra y los otros cinco pines se han de conectar a cinco entradas digitales de la placa Arduino. Cada uno de estos pines se corresponde con una posición del pulsador (arriba, abajo, derecha, izquierda, centro), y cuando el pulsador se coloque en una de ellas, el pin correspondiente recibirá una señal LOW (debido a las resistencias "pull-up" internas de la plaquita).

Cambiar el voltaje de referencia de las lecturas analógicas

Ya comentamos en el capítulo 2 el concepto de "voltaje de referencia". Básicamente, es un valor que define la precisión con la que se convierte una señal analógica (leída en una entrada analógica con *analogRead()*) a una señal digital equivalente (transformada por el conversor interno de la placa Arduino). Cuanto menor sea ese voltaje de referencia, mayor es la exactitud en la conversión. Por defecto, este valor es de 5V, pero si la precisión que nos ofrece no nos es suficiente (ya que con 5V de referencia se llega a una distancia entre valores digitales contiguos

de aproximadamente 4,9mV), lo podemos reducir haciendo uso de la función *analogReference()* y, opcionalmente, del pin AREF de la placa.

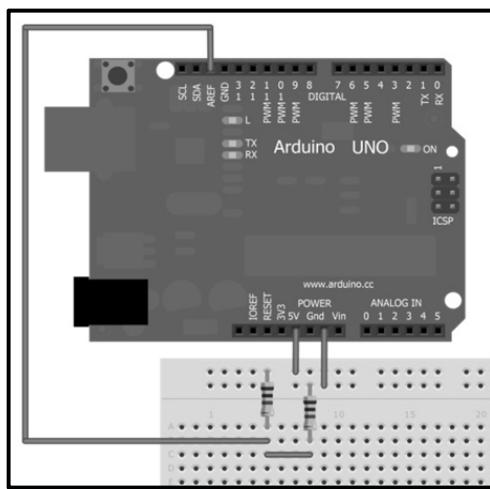
analogReference(): configura el voltaje de referencia usado para la conversión interna de valores analógicos en digitales. Dispone de un único parámetro, que en la placa Arduino UNO puede tener los siguientes valores: la constante predefinida DEFAULT (que equivale a establecer el voltaje de referencia es 5V –o 3,3V en las placas que trabajen a esa tensión–, el cual es el valor por defecto), o la constante predefinida INTERNAL (donde el voltaje de referencia entonces es de 1,1V) o la constante predefinida EXTERNAL (donde el voltaje de referencia entonces es el voltaje que se aplique al pin AREF –"Analogue REference"–). Es muy importante ejecutar siempre esta función antes de cualquier lectura realizada con *analogRead()* para evitar daños en la placa. Esta función no tiene valor de retorno.

Si usamos la opción de utilizar una fuente de alimentación externa para establecer el voltaje de referencia, deberemos conectar el borne positivo de esa fuente al pin-hembra AREF de la placa y su borne negativo a la tierra común. Para no dañar la placa, es muy importante que el voltaje de referencia externo aportado al pin AREF no sea nunca menor de 0V (es decir, invertido de polaridad) ni mayor de 5V.

Rebajar el voltaje de referencia es útil cuando sabemos que los valores de las entradas analógicas no alcanzan nunca los 5V. Por ejemplo, si sabemos que el valor máximo de una determinada señal de entrada es de 2,5V, podríamos rebajar con una fuente adecuada el voltaje de referencia a 2,5V, para hacer así que cada valor digital se correspondiera con su valor analógico respectivo en una precisión de $2,5V/1024 \approx 2,5mV$, doblando pues así la resolución utilizada. Hay que tener en cuenta, no obstante, que los efectos del ruido aumentan a medida que aumentemos la resolución en los valores leídos.

Podemos construir una fuente de referencia externa fácilmente (y muy barata!) con un simple divisor de tensión. Por ejemplo, si queremos que el voltaje de referencia sea de 2,5V, podemos utilizar los 5V que aporta el pin "5V" de la propia placa Arduino para reducirlo mediante dos resistencias idénticas conectadas en serie (no importa su valor numérico concreto, lo que importa es que sean iguales) y utilizar el voltaje presente entre las dos (reducido a 2,5V por simple Ley de Ohm) como entrada del pin AREF. Es decir, diseñar un circuito como este:

EL MUNDO GENUINO-ARDUINO



Se deberían usar resistencias de baja tolerancia (1%) para que la señal de referencia sea lo suficientemente precisa. Aunque la manera más exacta de proporcionar una señal rebajada y estable sería usando un regulador de tensión apropiado.

El siguiente código muestra cómo utilizar un voltaje de referencia externo. Necesitamos un circuito con un potenciómetro, la patilla central que ha de estar conectada al pin analógico de entrada nº 2. Supondremos que la fuente externa de referencia proporciona por ejemplo 1,8V (y por tanto, dividiendo entre 1024 tendríamos 1,75 milivoltios de resolución entre cada lectura de *analogRead()*).

Ejemplo 6.43

```
int valorPot=0;
float voltajePot=0;
void setup(){
analogReference(EXTERNAL); //Ésta es la línea clave
Serial.begin(9600);
}
void loop(){
    valorPot=analogRead(2);
    //Primero se muestra la lectura en forma de porcentaje
    Serial.println((voltajePot/1024)*100);
/*La siguiente línea convierte el valor de valorPot en un valor de voltaje. Fijarse que los valores obtenidos de analogRead() van desde 0 a 1023 y los valores que queremos van desde 0 al voltaje de referencia (1,8V hemos supuesto en este ejemplo). Podríamos pensar en utilizar la función map(), pero
```

esta solo devuelve valores enteros, por lo que no nos serviría demasiado. No obstante, como en este caso particular los mínimos de ambos rangos son 0, en vez de map() podemos escribir una simple regla de proporcionalidad. Hay que tener en cuenta el detalle de haber especificado los valores numéricos de la fórmula como de tipo "float" (añadiéndoles el 0 decimal) para que el resultado obtenido sea de tipo "float" también y no se trunque.*/

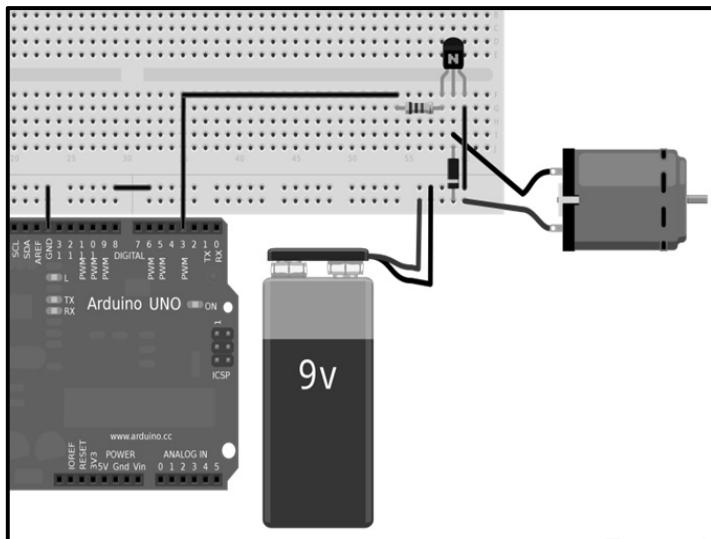
```

voltajePot=valorPot*(1.8/1023.0); //En V
voltajePot=voltajePot*1000; //Lo convertimos a mV
Serial.println(voltajePot);
delay(100);
}

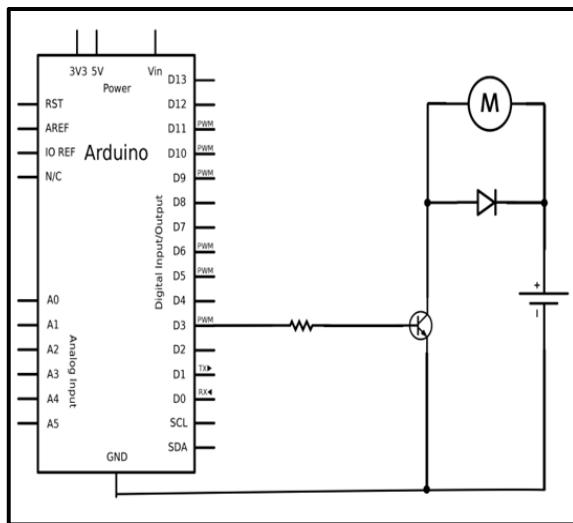
```

CONTROL DE MOTORES DC

En el apartado del capítulo anterior correspondiente a los motores no hablamos de cómo escribir código para controlar motores DC. Esto es porque no existe una librería Arduino específica para ellos, ya que estos dispositivos se pueden controlar con simples señales analógicas. Cuanto mayor sea el valor de la salida PWM enviada al motor, a más voltaje estará sometido y, por tanto, más rápido girará. Así de simple. Probaremos este funcionamiento con un circuito tal como el siguiente:



Tal vez su esquema eléctrico sea más claro:



Lo primero que llama la atención es la necesidad de utilizar una alimentación externa (en este caso, una pila de 9V) porque los motores por lo general realizan un elevado consumo, y con los 40 mA y 5V que ofrecen los pines de la placa Arduino no es suficiente. Lo segundo que llama la atención es el transistor. En este circuito lo utilizamos como simple válvula accionada por la señal PWM proveniente de la placa. Si esta vale 0, el transistor mantendrá abierto el circuito del motor y este no girará. A medida que se vaya aumentando el valor de voltaje PWM enviado a la base del transistor, por esta circulará más corriente. Esto provocará que entre el colector y el emisor vaya pasando más y más corriente (o dicho de otra forma, que se vaya reduciendo la resistencia entre estos dos puntos más y más). Esto provocará a su vez que el motor vaya siendo sometido a un mayor potencial y por tanto, que vaya girando más rápido. El caso extremo aparece cuando el transistor actúa como un simple circuito cerrado; esto ocurre al llegar a un determinado valor de la señal PWM, el cual provoca la llamada "intensidad de saturación" en la base del transistor. En ese momento, la resistencia entre colector y emisor es nula, por lo que el motor se somete a la tensión íntegra ofrecida por la fuente, y, por tanto, el motor gira a la máxima velocidad.

Está claro que si en vez de haber sometido el transistor a una señal PWM, lo sometiéramos a una señal digital, el transistor actuaría como un interruptor, parando o moviendo el motor según si recibe una señal LOW o HIGH, respectivamente.

Es importante aclarar que en el esquema hemos utilizado un transistor NPN de tipo Darlington modelo TIP120 (otro similar sería el BD645). El TIP120 puede

trabajar sometido a tensiones entre el colector y el emisor (V_{ce}) de hasta 60V pero admite una corriente por el colector (I_c) de hasta tan solo 5A. Por otro lado, hay que tener en cuenta la ubicación de sus patillas para comprender el diseño del circuito: mirándolo de frente, la base es su patilla izquierda, el colector es su patilla central y el emisor es su patilla derecha; en otro modelo de transistor las patillas pueden estar intercambiadas. A destacar también el divisor de tensión (2KΩ sería un buen valor) conectado en serie a su base.

El mismo circuito lo podríamos haber diseñado utilizando un transistor de tipo MOSFET (por ejemplo, el RFP30N06LE). En ese caso, la única diferencia es sustituir la resistencia en serie del circuito anterior por una resistencia "pull-up" (de 10KΩ por ejemplo) situada entre el pin de salida PWM de la placa Arduino y tierra. Y ya está. La ventaja de utilizar este transistor es que, aunque su V_{ce} máximo admitido sigue siendo de 60V, la I_c máxima es de 30A, pudiendo ser usado por tanto en circuitos con mucho mayor consumo. Podemos utilizar otros transistores diferentes (incluso un optoacoplador), pero en todo caso deberemos comprobar en su datasheet que sus V_{ce} e I_c máximos sean valores dentro del rango de trabajo del circuito.

Por otro lado, la resistencia que conectemos en serie a la base del transistor (necesaria para no dañarla) debe ser de un valor lo suficientemente pequeño como para poder alcanzar la corriente mínima con la que se llega al estado de saturación del transistor. El valor adecuado de esta resistencia se puede calcular mediante la expresión $(V_{pin} - 0,7)/I_{bsat}$, donde V_{pin} es la tensión que proporciona el pin de la placa Arduino donde está conectada la base (en nuestro caso, siempre será 5V), 0,7V es la caída de tensión típica que existe permanentemente entre la base y el emisor de un transistor NPN (aunque se puede mirar su valor exacto en el datasheet del transistor concreto buscando un dato llamado V_{be} –o también V_{th} , V_y o V_d –) e $I_{b(sat)}$ es precisamente la corriente mínima con la que se llega al estado de saturación del transistor, más allá de la cual no se obtiene más amplificación en I_c . Si $I_{b(sat)}$ no nos lo proporcionaran en el datasheet, se podría calcular fácilmente a partir de la expresión $I_{b(sat)} = I_{c(max)}/h_{FE}$, donde tanto $I_{c(max)}$ como h_{FE} sí que son consultables en el datasheet (h_{FE} es la llamada ganancia de corriente del transistor: si hay varios valores se ha de elegir el más pequeño).

NOTA: Es importante no aplicar intensidades mucho más grandes que $I_{b(sat)}$ para no quemar el transistor.

Lo importante del circuito anterior es fijarse en que hemos utilizado un transistor para "desacoplar" dos circuitos que funcionan a diferentes voltajes para que no haya peligro de dañarlos. Es decir, los 9V utilizados para alimentar el motor no

EL MUNDO GENUINO-ARDUINO

son nunca percibidos por la placa Arduino, la cual continúa trabajando como siempre a 5V. El transistor hace pues de barrera entre los dos circuitos, de tal forma que uno (sometido a 5V) simplemente sirve para controlar el funcionamiento del otro (sometido a 9V) pero sin peligro para el primero. Si se desea una protección algo más sofisticada, se puede sustituir el transistor por un optoacoplador, pero en el circuito anterior esto no es necesario.

Es interesante conocer la potencia consumida por el transistor en un circuito como este, calculable mediante la expresión $P=I_c \cdot V_{CE}$. En el modo de corte, $I_c = 0$, por lo que la potencia consumida es 0, y en el modo de saturación $V_{CE} = 0$ (aproximadamente), por lo que la potencia también es prácticamente nula. Esto quiere decir que el transistor no debería calentarse en ningún rango de su uso y no se necesita tener en cuenta ninguna potencia máxima de trabajo.

Por último, la presencia del diodo tiene una razón: hace de diodo "fly-back". Ya hemos comentado en el capítulo anterior que cuando se deja de alimentar una bobina (y los motores están hechos de bobinas) durante unos milisegundos entre los bornes del motor aparece un voltaje de hasta varios centenares de voltios polarizado a la inversa del de la fuente. Cuando hay alimentación, el diodo está situado "al revés" y no hace nada, pero cuando aparece ese voltaje inverso, el diodo permite que la corriente generada retorne al motor y no se dirija al transistor (ya que si no, lo freiría). Necesitamos por tanto un diodo lo suficientemente rápido para reaccionar al voltaje inverso y lo suficientemente fuerte para resistirlo: el modelo 1N4001 o SB560 son buenas opciones.

Un error común en el principiante es olvidarse de conectar el emisor del transistor, además de al polo negativo de la fuente, al pin GND de la placa. Esta conexión es necesaria porque la señal PWM necesita un camino de retorno.

Otro detalle que debemos tener en cuenta es que aunque podemos considerar que el montaje anterior se compone de dos circuitos separados por el transistor (el circuito del motor y el circuito de la señal PWM), estos están interrelacionados, por lo que es fundamental que las tierras de ambos sean comunes. Es por eso que el polo negativo de la fuente está conectado al pin GND de la placa Arduino.

Aquí vemos un sketch que aumenta la velocidad de giro del motor hasta llegar a su máximo y seguidamente la disminuye hasta pararse, y volver a acelerarse otra vez, y así siempre:

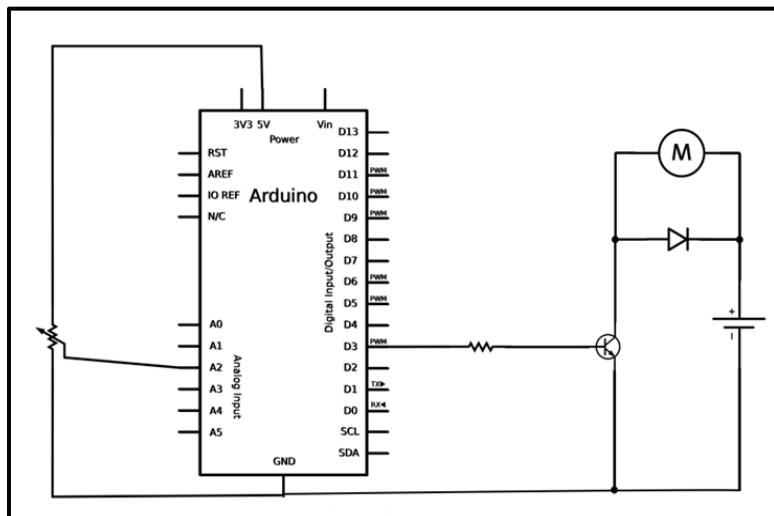
Ejemplo 6.44

```

void setup(){
  pinMode(3, OUTPUT);
}
void loop(){
  int i;
  for(i = 0; i<255; i++){
    analogWrite(3,i);
    delay(15);
  }
  for(i = 255; i>=0; i--){
    analogWrite(3,i);
    delay(15);
  }
}

```

Añadiremos ahora al circuito anterior un potenciómetro (en el pin de entrada analógica nº 2, por ejemplo), de tal manera que la velocidad de giro del motor venga ahora dada por el valor leído de la resistencia variable. El esquema eléctrico del nuevo diseño ha de ser así:



Para hacer el sketch algo más sofisticado, hemos añadido la condición de que si el valor leído por el potenciómetro no llega a un nivel mínimo ("umbral"), el motor no se pone en marcha; y cuando se pone en marcha, entonces ya sí que su velocidad es proporcional a la lectura recibida del potenciómetro:

Ejemplo 6.45

```

int lectura = 0;
int umbral= 700;
void setup() {
    pinMode(3, OUTPUT);
}
void loop() {
    lectura = analogRead(2);
    if(lectura > umbral) {
        //Ajusto el valor de "lectura" para que caiga entre 0 y 255
        analogWrite(3, lectura/4);
    } else {
        digitalWrite(3, LOW);
    }
}

```

El código anterior es perfectamente válido para cualquier otro tipo de entrada analógica, como las que veremos en el capítulo siguiente. Así, un termistor o un fotoresistor se comportan de forma equivalente a un potenciómetro, por lo que podríamos diseñar circuitos que pusieran en marcha un motor cuando se detectara cierta cantidad de luz (para por ejemplo, bajar una persiana) o cuando se detectara una cierta temperatura (para por ejemplo activar un ventilador), etc. Por otro lado, no debería costar demasiado sustituir el potenciómetro del diseño anterior por un pulsador, y conseguir que el motor solamente gire cuando se mantenga apretado el pulsador. Se deja como ejercicio.

Otro ejercicio podría ser probar de enviar, tal como ya hicimos con los servomotores, un dato numérico a través del canal serie para, en este caso, especificar la velocidad a la que deseamos girar el motor. Se deja como ejercicio.

El chip L293

Desgraciadamente, los circuitos con un solo transistor ya sabemos qué carencia tienen: el motor solo puede girar en un sentido. Para que pueda girar en los dos, se necesita utilizar un "puente H". Podemos optar por utilizar en nuestros circuitos un chip integrado como el L293, el SN754410 o el L298, conectándolos directamente a una breadboard, por ejemplo. Todos estos chips permiten controlar 2 motores DC conectando 4 salidas digitales de nuestra placa (2 para cada motor). Para hacer girar cada motor en un determinado sentido, se ha de establecer una salida a HIGH y la otra a LOW y para hacerlo girar en sentido contrario, se han de establecer al revés (LOW y HIGH). El L293 y el SN754410 tienen una distribución de patillas idéntica, pero diferente al L298. En nuestro ejemplo utilizaremos el L293, ya que su distribución es algo más sencilla.

Este chip en realidad tiene dos puentes (uno para cada motor): uno en su lado izquierdo y otro en el derecho. Puede entregar hasta 1A por motor y opera entre 4,5V y 36V, así que hay que elegir un motor DC acorde con estas características. La distribución de sus pines es la siguiente:

- Pin 1:** activa o desactiva un motor, según si recibe una señal HIGH o LOW. También sirve para especificar la velocidad de giro si recibe una señal PWM.
- Pin 2:** envía la señal de giro (HIGH o LOW) en un sentido para un motor.
- Pin 3:** donde se conecta uno de los dos terminales de un motor.
- Pin 4 y 5:** tierra.
- Pin 6:** donde se conecta el otro terminal de un motor.
- Pin 7:** envía la señal de giro (HIGH o LOW) en el otro sentido para un motor.
- Pin 8:** alimentación del motor (debe ser un valor suficiente).
- Pin 9-11:** si no se usa un segundo motor, pueden estar desconectados. El nº 9 es para enviar la señal de activación/desactivación y PWM, el nº 10 es para enviar la señal de giro en un sentido y el nº 11 es para conectar un terminal del segundo motor.
- Pin 12 y 13:** tierra.
- Pin 14 y 15:** si no se usa un segundo motor, pueden estar desconectados. El nº 14 es para conectar el otro terminal del segundo motor y el nº 15 es para enviar la señal de giro en sentido contrario.
- Pin 16:** alimentación del propio chip (ha de estar conectado a 5V).

Según las combinaciones de señales que se envíen a las diferentes patillas de este chip, el motor girará más o menos velozmente en un determinado sentido. Concretamente, si suponemos que el pin nº 1 envía una señal HIGH, cuando en el pin nº 2 haya una señal HIGH y en el pin nº 7 haya una señal LOW, el motor girará a la derecha, y cuando en el pin nº 2 haya una señal LOW y en el pin nº 7 haya una señal HIGH, el motor girará a la izquierda. Si en ambos pines hay una señal igual (HIGH o LOW), el motor se parará.

Al circuito formado por la placa Arduino, el motor y el chip L293 es recomendable añadir además un condensador de 10-100 µF conectado cerca del motor, entre la alimentación del motor (pin nº 8) y tierra. Esto suavizará los picos de voltaje que se producen cuando el motor se enciende y evitará que el microcontrolador se reinicie debido a ello: es decir, estamos hablando de un condensador "by-pass". Cuanto mayor sea la capacidad del condensador, más carga podrá almacenar pero más tiempo necesitará para liberarla.

Podemos incluir también un pulsador conectado a algún pin digital de la placa Arduino para alternar los dos sentidos de giro del motor. Lo más sencillo es hacer

EL MUNDO GENUINO-ARDUINO

girar el motor en un sentido mientras el pulsador no está apretado, y hacerlo girar en el otro cuando sí está apretado. Así es como el sketch 6.44 opera, de hecho. Una vez ya tenemos el circuito montado, el código es simple:

Ejemplo 6.46

```
const int switchPin = 2; //Conectado al pulsador
const int motor1Pin = 3; //Conectado al pin nº 2 del L293
const int motor2Pin = 4; //Conectado al pin nº 7 del L293
const int enablePin = 9; //Conectado al pin nº 1 del L293
void setup() {
    pinMode(switchPin, INPUT);
    pinMode(motor1Pin, OUTPUT);
    pinMode(motor2Pin, OUTPUT);
    pinMode(enablePin, OUTPUT);
    digitalWrite(enablePin, HIGH);
}
void loop() {
    if (digitalRead(switchPin) == HIGH) {
        digitalWrite(motor1Pin, LOW);
        digitalWrite(motor2Pin, HIGH);
    } else {
        digitalWrite(motor1Pin, HIGH);
        digitalWrite(motor2Pin, LOW);
    }
}
```

En el código anterior no podemos variar la velocidad de giro del motor porque el pin nº 1 del L293D tan solo recibe una señal HIGH y ya está. Pero podríamos enviarle una señal de tipo PWM mediante *analogWrite()* y hacerla variar por la acción de un potenciómetro, por ejemplo. Así podríamos controlar la velocidad de giro también. Se deja como ejercicio.

Módulos de control para motores DC

Otra opción diferente para controlar motores DC, en vez de conectar el chip directamente a una breadboard, es utilizar alguna placa breakout específica. De esta forma, el cableado es menos enrevesado y el montaje es mucho más sencillo. Usando un módulo, lo único que deberemos hacer es, además de alimentarlo y enchufar el motor a sus bornes pertinentes, localizar los dos terminales de control de sentido de giro (y conectarlos a dos pines digitales de la placa Arduino), y el terminal de activación/desactivación y control de la velocidad de giro (y conectarlo a un pin de salida PWM). Y poco más.

La placa TB6612FNG

Una placa breakout recomendable es por ejemplo el producto nº 9457 de Sparkfun. Este módulo incorpora el chip TB6612FNG, el cual incorpora dos "puentes H", por lo que es capaz de manejar dos motores DC de forma independiente, soportando un consumo constante por motor de 1,2A (con picos puntuales de 3,2A) y de hasta 13V de trabajo. Las conexiones a realizar son las siguientes:

Placa breakout	Exterior
VM	Borne positivo de la fuente de alimentación externa
VCC	Pin "5V" de Arduino.
GND	Borne negativo de la fuente de alimentación externa y Pin GND de Arduino
A01	Terminal del motor A
A02	Terminal del motor A
B02	Terminal del motor B
B01	Terminal del motor B
GND	-
PWMA	Pin de salida PWM de Arduino
AIN2	Pin de salida digital de Arduino
AIN1	Pin de salida digital de Arduino
STBY	Pin de salida digital de Arduino
BIN1	Pin de salida digital de Arduino
BIN2	Pin de salida digital de Arduino
PWMB	Pin de salida PWM de Arduino
GND	-

Los pines "IN1" y "IN2" sirven para controlar el sentido de giro de los motores A y B según el caso: cuando "IN1" esté a HIGH e "IN2" a LOW, el motor girará en un sentido e, intercambiando sus valores, girará en el otro. El pin "PWM" sirve para controlar la velocidad de los motores A y B, según el caso. El pin "STBY" permite desconectar ambos motores de la fuente de alimentación si se le envía una señal de valor LOW. A continuación, se muestra un código que controla solo un motor:

Ejemplo 6.47

```
int STBY = 10; //Salida digital donde se conecta el pin "STBY"  
//Motor A: un pin controla la velocidad y otros dos el sentido  
int PWMA = 3; //Salida PWM donde se conecta el pin "PWMA"  
int AIN1 = 9; //Salida digital donde se conecta el pin "AIN1"  
int AIN2 = 8; //Salida digital donde se conecta el pin "AIN2"  
void setup(){  
    pinMode(STBY, OUTPUT);  
    pinMode(PWMA, OUTPUT);  
    pinMode(AIN1, OUTPUT);  
    pinMode(AIN2, OUTPUT);  
}  
void loop(){  
    //Muevo el motor 1 a máxima velocidad a la izquierda  
    mover(0, 255, 1);  
    delay(1000);  
  
    parar();  
    delay(250);  
    //Muevo el motor 1 a media velocidad a la derecha  
    mover(0, 128, 0);  
    delay(1000);  
    parar();  
    delay(250);  
}  
void mover(int motor, int velocidad, int direccion){  
    //motor: vale 0 para el motor A y 1 para el motor B  
    //velocidad: 0 equivale a ninguna y 255 a la velocidad máxima  
    //dirección: 0 es el sentido de las agujas del reloj y 1 al revés  
    boolean inPin1 = LOW; //Asumo un sentido de giro inicial  
    boolean inPin2 = HIGH;  
    digitalWrite(STBY, HIGH); //Desactivo el standby  
    if(direccion == 0){  
        inPin1 = LOW;  
        inPin2 = HIGH;  
    }  
    if(direccion == 1){  
        inPin1 = HIGH;  
        inPin2 = LOW;  
    }  
    if(motor == 0){  
        digitalWrite(AIN1, inPin1);  
        digitalWrite(AIN2, inPin2);  
    }
```

```

        analogWrite(PWMA, velocidad);
    }
}

void parar(){
    digitalWrite(STBY, LOW); //Activo el standby
}

```

Otros módulos

Otro módulo diferente es el "Motor Driver 2A Dual L298 H-Bridge" de Sparkfun (producto nº 9670), que contiene el chip L298, permitiendo conectar dos motores DC independientes a 2A, o bien un motor DC a 4A. Otras placas breakout similares son el "L298 Compact Motor Driver" de Solarbotics, o el "DC Motor Driver Breakout" de CuteDigi o el "Two-Channel DC Motor Driver Breakout Board" de SCMDigi o el "3-Wire Robot Motor Driver Board" de Cal-Eng. Todos estos módulos disponen del chip L298 y de dos entradas digitales para controlar el sentido de giro y el freno de cada motor más otra entrada PWM más para controlar la velocidad. Dependiendo de la ubicación de un puente de plástico específico (como los ofrecidos por DFRobot con código **FIT0140**, también llamados "jumpers"), pueden usar la misma alimentación para los motores y el chip (convenientemente regulada a 5V), o bien recibir la alimentación de parte de dos fuentes externas separadas.

Cal-Eng por su parte, también ha diseñado las placas NanoDuino y Microduino, que no es más que una placa Arduino con el chip L298 integrado, a un reducido tamaño, y programable vía FTDI. Esta placa está especialmente pensada para robótica.

Shields de control para motores DC (y paso a paso)

En vez de los módulos externos vistos en el apartado anterior podemos utilizar shields específicos para el control de motores DC (los cuales suelen servir también para el control de steppers y servomotores). De esta forma, podemos ahorrar cableado y ganar espacio en nuestros proyectos: tan solo deberemos alimentar el shield con una fuente externa, conectar los terminales del motor a los bornes adecuados y nada más.

El "Adafruit Motor Shield"

Además del shield oficial de Arduino (que, recordemos, permite manejar dos motores DC o bien un motor paso a paso), un shield interesante a considerar es el "Motor/Stepper/Servo Shield", producto nº 81 de Adafruit. Este shield tiene dos conectores para sendos servomotores (a 5V) e incorpora además 2 chips L293D. Esto

EL MUNDO GENUINO-ARDUINO

quiere decir que tiene 4 "puentes H" (cada chip L293D tiene dos), por lo que puede controlar hasta 4 motores DC bidireccionales o bien 2 motores paso a paso unipolares o bipolares (o bien una combinación de 2 motores DC y 1 motor paso a paso). Para conectar un motor DC, simplemente se deberían conectar sus dos cables a algunos de los terminales M1, M2, M3 y M4; para conectar un motor stepper bipolar, los cables de una bobina se deberían conectar al terminal M1 y los de la otra a M2 (un segundo stepper bipolar se podría conectar a los terminales M3 y M4); si el stepper fuera de tipo unipolar, se conectaría igual y el resto de cables sobrantes tendrían que ir a tierra.

Todos los pines digitales de entrada/salida (excepto el nº 2) de este shield son utilizados para la comunicación entre la placa Arduino y shield, por lo que no se podrán utilizar para otra cosa. Las 6 entradas analógicas sí que están disponibles (y se pueden usar como pines digitales también).

El chip L293D de este shield proporciona 0,6A por puente, con un pico máximo de 1,2A para momentos puntuales, y puede manejar motores funcionando entre 4,5V y 25V. Para alimentar los motores DC y paso a paso, se necesita una fuente externa conectada al bloque de dos bornes etiquetado como "EXT_PWR", y dependiendo de si está colocado o no un jumper concreto en el shield, esta alimentación servirá también para la placa Arduino o esta deberá alimentarse separadamente vía USB u otra fuente externa independiente (que es lo recomendable).

Para manejar los servomotores conectados a este shield se puede utilizar la librería oficial Servo de Arduino, pero para controlar los motores DC y paso a paso, es necesario utilizar una librería propia de Adafruit, la "Adafruit Motor Shield Library" (<https://github.com/adafruit/Adafruit-Motor-Shield-library>). Desgraciadamente, no disponemos de espacio suficiente para detallar el funcionamiento de esta librería, por lo que remito a los códigos de ejemplo y a su documentación oficial (<http://www.ladyada.net/make/mshield/use.html>).

Existe otra librería suplementaria a la "Adafruit Motor Shield Library" que aporta un control extra avanzado en el uso de motores paso a paso, con posibilidad de aceleración y desaceleración, o con acceso concurrente a varios steppers. Esta librería se puede descargar de <https://github.com/adafruit/AccelStepper>, donde también se pueden consultar varios códigos de ejemplo.

En realidad, si se desea utilizar este shield tan solo para manejar motores DC, es posible no tener que utilizar la librería de Adafruit y simplemente escribir código

Arduino tal cual, si se siguen las instrucciones especificadas en la siguiente página de la web oficial: <http://arduino.cc/playground/Main/AdafruitMotorShield>.

Otros shields

Existen más shields que incorporan la circuitería necesaria para manejar distintos tipos de motores (DC, Servo o Steppers). Por ejemplo, un shield prácticamente idéntico al de Adafruit es el "**Rotoshield**" de Snootlab: utiliza el mismo chip L293D, permite controlar el mismo número y tipo de motores y trabaja en los mismos rangos de consumo. Entre las diferencias podemos destacar que utiliza comunicación I²C con la placa Arduino (por lo que libera muchos más pines de entrada/salida para poderlos utilizar en otros menesteres), que ofrece hasta ocho salidas PWM extras, y que acepta una alimentación externa de hasta 18V. Para programarla se necesita utilizar la librería llamada "**Snootor**", disponible en <https://github.com/Snootlab/Snootor>.

Un shield que utiliza el chip L298N (como el del shield oficial de Arduino) es el "**Motomama**" de IteadStudio. Con él se puede manejar hasta 2 motores DC (con un consumo de hasta 2A cada uno) o un motor paso a paso. Como los anteriores shields, es recomendable que se alimente mediante una fuente externa conectada a los bornes adecuados del mismo, la cual puede alimentar a su vez a la placa Arduino o no, según la colocación de un jumper específico. También puede ser alimentada si hay una fuente externa conectada directamente al zócalo "jack" de la placa Arduino. Como mayor novedad, este shield aporta un zócalo XBee para poder insertar un módulo de este tipo y así poder controlar el shield inalámbricamente. Esto está especialmente pensado para la construcción de vehículos teledirigidos. No utiliza ninguna librería propia para el manejo de los motores.

Otro shield que no necesita ninguna librería de terceros para ser utilizado es el shield "**Ardumoto**" de Sparkfun, el cual también está basado en el chip L298 (así que también es capaz de controlar 2 motores DC o un motor paso a paso). La alimentación de los motores se obtiene a través del pin "Vin" de la placa Arduino, por lo que es allí donde deberíamos conectar la fuente de alimentación externa. Fuentes adecuadas para este shield son por ejemplo el producto nº 298 de Sparkfun (un adaptador AC/DC de 9V), o bien los productos nº 9703 y 9704 (baterías LiPo). Este shield también incorpora un regulador interno que, a partir del voltaje de "Vin", ofrece una tensión de 5V para proteger su circuitería más delicada.

El shield "Ardumoto" dispone de los bornes "OUT1" y "OUT2" para conectar los terminales de un motor DC, y los bornes "OUT3" y "OUT4" para conectar los del otro. A partir de aquí, para especificar el sentido de giro del primer motor deberemos

EL MUNDO GENUINO-ARDUINO

utilizar en nuestros sketches el pin digital nº 12, ya que este es el pin conectado internamente a la patilla correspondiente del L298; según se envíe por ese pin un valor HIGH o LOW el motor girará en un sentido o en otro. Observar que tan solo se requiere una única conexión para especificar el sentido de giro (en vez de dos como hemos visto en hardware anterior). Igualmente, para especificar la velocidad de giro de ese primer motor deberemos utilizar en nuestros sketches el pin PWM nº 3. Si trabajamos con un segundo motor el sentido de giro se manipula mediante el pin digital nº 13 y la velocidad de giro mediante el pin PWM nº 11. Como muestra, he aquí este código, el cual maneja un solo motor:

Ejemplo 6.48

```
/*En esta placa, el motor "A" es el conectado a los terminales 1 y 2, y el motor "B" es el conectado a los terminales 3 y 4*/
int pwm_a = 3; //Pin de control de velocidad del motor A
int dir_a = 12; //Pin de control de sentido de giro del motor
int i = 0;
void setup(){
    pinMode(pwm_a, OUTPUT);
    pinMode(dir_a, OUTPUT);
    analogWrite(pwm_a, 100);
}
void loop(){
/*La siguiente orden equivale a establecer la patilla nº 1 del chip para el control de sentido de giro a HIGH y la patilla nº 2 a LOW, pero no gira el motor hasta que no se le aplique una señal PWM */
    digitalWrite(dir_a, HIGH);
    fadein(); delay(1000);
    fadeout(); delay(1000);
    //Paro el motor
    analogWrite(pwm_a, 0); delay(2000);
/*La siguiente orden equivale a establecer la patilla nº 1 del chip para el control de sentido de giro a LOW y la patilla nº 2 a HIGH, pero no gira el motor hasta que no se le aplique una señal PWM */
    digitalWrite(dir_a, LOW);
    fadein(); delay(1000);
    fadeout(); delay(1000);
    //Paro el motor
    analogWrite(pwm_a, 0); delay(2000);
}
void fadein(){
    for(i = 0 ; i <= 255; i+=5) {
        analogWrite(pwm_a, i);
        delay(30); //Me espero para ver el resultado
    }
}
```

```

void fadeout(){
    for(i = 255 ; i >= 0; i -=5) {
        analogWrite(pwm_a, i); delay(30);
    }
}

```

Sparkfun distribuye otro shield llamado "**Monster Motor Shield**", que es una versión modificada del "Ardumoto" donde se ha sustituido el chip L298 por un par de chips VNH2SP30 y se ha reforzado la circuitería interna para soportar voltajes más elevados.

Otro shield que incorpora el chip L298P es el "**2A Motor shield**" de DFRobot. Como su nombre indica, puede ofrecer hasta un máximo de 2A de corriente a los dos motores DC que se le conecten. Puede ser alimentado a través de la fuente conectada al conector "jack" de la placa Arduino (funcionando entonces a un voltaje entre 7 y 12V), o bien de forma independiente de una fuente externa (funcionando entonces a un voltaje entre 5 y 35V); para cambiar de un tipo a otro de alimentación se ha de modificar la ubicación de un jumper determinado del shield. El control del motor DC conectado a los terminales "M1+" y "M1-" se efectúa mediante el pin digital nº 4 (para especificar el sentido de giro) y el pin PWM nº 5 (para especificar la velocidad de giro). El control del motor DC conectado a los terminales "M2+" y "M2-" se efectúa mediante el pin digital nº 7 y el pin PWM nº 6. DFRobot también distribuye otro shield llamado "**1A Motor Shield**" que viene con el chip L293, por lo que solo permite un máximo de 1A de corriente para ser consumida por los motores conectados.

Otro shield muy similar a los anteriores es el "**Motor shield**" de Open-Electronics, el cual incorpora el chip L298P, por lo que también puede manejar tanto motores DC como paso a paso con una corriente máxima de salida de 2A por cada canal de salida. La alimentación puede ser externa o proveerla la propia placa Arduino.

En otra escala trabajan los shields Megamoto y Megamoto Plus de Robot Power, los cuales permiten controlar un motor DC en ambos sentidos o bien dos motores DC en el mismo sentido de giro con un consumo máximo de hasta 40A y 25V (es decir, hasta ¡1100W!).

Un shield específico para la conexión y uso de hasta 16 servomotores con solo la utilización de 4 pines digitales (del nº 6 al 9) es el "**Renbotics ServoShield**" de Seeedstudio. Además, dispone de una amplia zona para realizar prototipados. Se ha de programar con una librería propia, disponible en la página web del producto.

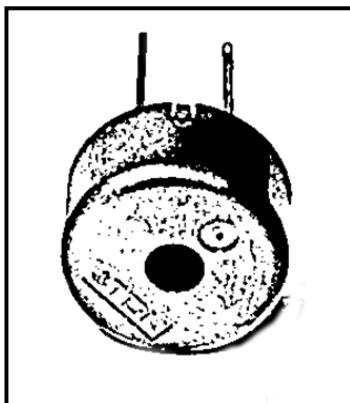
EL MUNDO GENUINO-ARDUINO

Un shield específico para la conexión y uso de hasta 2 motores paso a paso es el "**Arduino dual step motor driver shield**" de IteadStudio. Funciona con una alimentación de entre 5V y 30V y no requiere ninguna librería específica para ser programada. Se basa en el chip controlador de steppers A3967.

Un módulo –que no shield– especializado en el control de motores steppers muy parecido al shield anterior de IteadStudio (ya que está basado en el mismo chip A3967) es el llamado "**Easy Driver**" (comercializado entre otros por Sparkfun con código de producto 10267). Recomiendo la lectura del sencillo tutorial disponible en su web (<http://www.schmalzhaus.com>) para comprender de una manera rápida su funcionamiento y sus capacidades.

EMISIÓN DE SONIDO

Uso de zumbadores



Un zumbador piezoeléctrico (en inglés, "buzzer" o "piezobuzzer") es un dispositivo que consta internamente de un disco de metal, que se deforma (debido a un fenómeno llamado piezoelectricidad) cuando se le aplica corriente eléctrica. Lo interesante es que si a este disco se le aplica una secuencia de pulsos eléctricos de una frecuencia suficientemente alta, el zumbador se deformará y volverá a recuperar su forma tan rápido que vibrará, y esa vibración generará una onda de sonido audible.

Recordemos que un sonido aparece cuando vibra alguna fuente y estas vibraciones son transmitidas en forma de onda a través de algún medio elástico (como el aire o el agua). Si la frecuencia de esa onda está dentro de un rango determinado (el llamado "espectro audible", el cual va de 20Hz a 20KHz), cuando llega al oído humano esas oscilaciones en la presión del aire son convertidas en el sonido que nuestro cerebro percibe.

Cuanto mayor sea la frecuencia de la onda sonora, más agudo será el sonido resultante (y al revés: cuanto menor es esa frecuencia, más grave es el sonido). Por tanto, para generar diferentes tonos con nuestro zumbador deberemos hacer vibrar la membrana a distintas frecuencias. Para ello, deberemos emitir desde nuestro Arduino pulsos digitales con la frecuencia deseada. Los zumbadores admiten voltajes

desde 3 a 30 voltios, así que los 5V de una salida digital de Arduino los soporta perfectamente.

Así pues, para hacer sonar un zumbador típico (como por ejemplo, el producto nº 160 de Adafruit o similar), simplemente deberemos conectar uno de sus terminales a tierra y el otro a un pin-hembra digital del Arduino configurado como salida digital. Atención, dependiendo del modelo de zumbador, este puede ser un componente polarizado o no, por lo que hay que fijarse si tiene alguna marca indicando la polaridad de sus terminales. Si se desea, al zumbador se le puede conectar en serie un divisor de tensión (de 100Ω está bien): pero hay que tener en cuenta que cuanta mayor resistencia tenga este divisor, menor será el volumen del sonido generado.

Una vez realizadas las conexiones, podemos ejecutar este sketch:

Ejemplo 6.49

```
void setup (){
    pinMode (8, OUTPUT); //Pin al que está conectado el zumbador
}
void loop (){
    digitalWrite (8, HIGH);
    delayMicroseconds (1000);
    digitalWrite (8, LOW);
    delayMicroseconds (1000);
}
```

Lo que estamos haciendo es enviar consecutivamente pulsos digitales HIGH y LOW a una velocidad tan elevada que hace vibrar el zumbador a una frecuencia audible por el ser humano. Concretamente, el pulso HIGH lo hacemos durar 1 ms y el pulso LOW otro 1ms, por lo que el periodo de esta onda cuadrada es de 2ms (0,002s). Por tanto, como $f=1/T$, la frecuencia de la señal audible generada será de $1/0,002 = 500\text{Hz}$.

Podemos hacer que la frecuencia del sonido vaya cambiando. Por ejemplo, el siguiente sketch reproduce (una sola vez) un sonido que pasa de ser grave a ser agudo de forma continua. Esto es porque al principio su periodo es de 10000ms (es decir, $1/0,01 = 100\text{Hz}$) y al final su periodo es de 100ms (es decir, $1/0,0001 = 10\text{KHz}$). Fijarse que podemos variar la duración del sonido si modificamos el incremento (es decir, el tercer parámetro del for, que en este caso es un decremento). Para probar este código, se ha de usar el mismo circuito que el ejemplo anterior.

EL MUNDO GENUINO-ARDUINO

Ejemplo 6.50

```
int tono = 10000; //Nota inicial. Es el periodo de la onda
void setup (){
    pinMode (8, OUTPUT);
    for (tono=10000; tono>=100; tono=tono-10){
        digitalWrite (8, HIGH);
        delayMicroseconds (tono/2);
        digitalWrite (8, LOW);
        delayMicroseconds (tono/2);
    }
}
void loop (){}
```

Al ejecutar el código anterior notarás que el zumbador permanece mucho rato emitiendo sonidos graves (parece que le cuesta "avanzar") para finalmente "acelerar" y emitir los sonidos más agudos en muy poco tiempo. Podemos mitigar este efecto, y hacer que el sonido transcurra desde los sonidos graves hasta los sonidos agudos de una forma más equilibrada realizando dos cambios en el código anterior. El primer cambio es convertir la variable "tono" en una variable de tipo "float". El segundo cambio es sustituir el decremento actual del bucle "for" (es decir, `tono=tono-10`) por otro tipo de decremento; concretamente, cambiar la resta por una división. Si hacemos esto (es decir, si escribimos por ejemplo `tono=tono*0.97`) controlaremos mejor la "velocidad" de la sirena. Podemos cambiar este factor 0,97 por otro, como 0,99 o 0,93 para observar qué ocurre.

Añadamos ahora al circuito anterior un potenciómetro. La patilla de un extremo (cualquiera) la conectaremos a la misma tierra que el zumbador, la patilla del otro extremo a la alimentación (que puede ser proporcionada por la propia placa Arduino a través del pin "5V") y la patilla central a un pin de entrada analógica de Arduino (por ejemplo, el nº 0). Gracias al sketch siguiente, seremos capaces de cambiar el tono del sonido generado por el zumbador simplemente girando la rueda del potenciómetro.

Ejemplo 6.51

```
int tono = 1000; //Nota inicial. Es el periodo de la onda
void setup (){
    pinMode (8, OUTPUT);
}
void loop (){
    digitalWrite (8, HIGH);
    delayMicroseconds (tono/2);
```

```

digitalWrite (8, LOW);
delayMicroseconds (tono/2);
tono=analogRead(0);
tono=map(tono,0,1023,1000,5000);
}

```

Lo que hacemos en el código anterior es obtener la lectura analógica del potenciómetro (que va de 0 a 1023) y utilizarla para definir el tiempo que duran los pulsos HIGH y LOW enviados al zumbador. Cuanto menos duren, más agudo será el sonido. Fijémonos que el valor leído primero se mapea para que esté dentro de un rango entre 1000 y 5000 (aunque perfectamente se puede elegir otro rango) y que la duración de los pulsos es "tono/2" porque el valor de "tono" representa el periodo total de la onda cuadrada (es decir, un pulso HIGH más un pulso LOW). Si hacemos los cálculos de párrafos anteriores, veremos que con el periodo de la onda cuadrada establecido entre 1000 y 5000, este sketch puede emitir sonidos entre 1KHz y 200Hz.

Si queremos variar el volumen del sonido emitido además de su frecuencia, deberemos de sustituir el divisor de tensión conectado en serie al zumbador por un segundo potenciómetro. Concretamente, deberíamos conectar un extremo de ese nuevo potenciómetro al pin-hembra digital del Arduino configurado como salida por donde se emite el sonido, el otro extremo del potenciómetro a tierra y su patilla central a un terminal del zumbador. De esta manera, podremos regular la intensidad de corriente recibida por el zumbador y, por tanto, el volumen del sonido emitido.

Otra manera de regular el volumen podría ser conectando el zumbador a una salida analógica de la placa en vez de a una digital. No obstante, de esta manera no podemos alterar la frecuencia de la onda emitida (ya que en un pin PWM esta es constante a 490Hz), por lo que el sonido que surge no es especialmente agradable.

Las funciones *tone()* y *noTone()*

Afortunadamente, normalmente no tendremos que utilizar *digitalWrite()* ni *analogWrite()* para emitir pitidos porque el lenguaje Arduino ofrece dos funciones especialmente pensadas para ello que nos facilitan mucho la escritura de nuestros sketches.

tone(): genera una onda cuadrada de una frecuencia determinada (especificada en Hz como segundo parámetro –de tipo "word"–) y la envía por el pin digital de salida especificado como primer parámetro, el cual deberá estar conectado algún tipo de zumbador o altavoz. Esta función no es

EL MUNDO GENUINO-ARDUINO

bloqueante; esto quiere decir que una vez comienza a emitirse el sonido, el sketch sigue su ejecución en la siguiente línea de código. La duración de la onda (en milisegundos) se puede especificar opcionalmente como tercer parámetro –su tipo es "unsigned long"– ; si no se indica, la onda se emitirá hasta que se llame a la función *noTone()*. La forma de la onda cuadrada no se puede alterar: siempre tiene un valor alto la mitad de su periodo y un valor bajo la otra mitad (lo que se llama tener un ciclo de trabajo del 50%); esto implica que no podremos alterar el volumen del sonido emitido (aunque, de hecho, con un 50% de ciclo de trabajo se emite al máximo volumen posible). En la documentación oficial se advierte que el uso de esta función puede desestabilizar la salida PWM de los pines 3 y 11 en la placa Arduino UNO. Esta función no retorna nada.

Solo una señal de audio puede generarse en un momento determinado: si ya hubiera una señal emitiéndose por otro pin diferente del especificado en *tone()* (porque tenemos más de un zumbador en nuestro circuito), la señal anterior continuaría emitiéndose como si nada y la nueva no se llegaría a emitir. Si esa señal previa estuviera emitiéndose en el mismo pin, entonces el efecto sí que sería la sustitución de la señal anterior por la nueva. Por lo tanto, si se desea emitir diferentes señales en múltiples pines, se deberá parar la emisión en uno (con *noTone()*) para empezar la emisión en otro.

noTone(): deja de generar la onda cuadrada que en principio estaba emitiéndose (por una ejecución previa de *tone()*) a través del pin especificado como (único) parámetro. Si no hay ninguna señal emitiéndose en ese pin, esta función no hace nada. No tiene valor de retorno.

Un sketch muy sencillo que muestra cómo trabajan estas funciones es el siguiente, y que se puede probar utilizando un circuito con solo un zumbador conectado a tierra y al pin digital de salida nº 8 de la placa Arduino (opcionalmente a través de un divisor de tensión). Al ejecutarlo se podrá escuchar un sonido ininterrumpido de sirena.

Ejemplo 6.52

```
int duracion = 250; //Duración del sonido
int freqmin = 2000; //La frecuencia más baja a emitir
int freqmax = 4000; //La frecuencia más alta a emitir
void setup(){
    pinMode(8, OUTPUT);
}
void loop(){
```

```

int i;
//Se incrementa el tono (se hace más agudo)
for (i = freqmin; i<=freqmax; i++){
    tone (8, i, duracion);
}
//Se disminuye el tono (se hace más grave)
for (i = freqmax; i>=freqmin; i--){
    tone (8, i, duracion);
}
}

```

Otro código más sofisticado es el siguiente (con el mismo circuito), en el cual se reproduce una melodía. Para lograrlo, el sketch emite, una tras otra y durante el tiempo preciso, las frecuencias exactas correspondientes a las notas musicales de esa melodía.

Ejemplo 6.53

```

//Frecuencias de las notas de la melodía
int melodía[] = {262, 196, 196, 220, 196, 247, 262};
/*Duración de las notas
(4 = dura un cuarto de tiempo, 8=dura una octava parte, etc)*/
int duraciónNota[] = {4,8,8,4,4,4,4,4};
void setup() {
    int i;
    //Re corro las 7 notas de la melodía una tras otra
    for (i = 0; i < 8; i++){
        /*Para calcular la duración de la nota, se divide un segundo por la cantidad marcada en duraciónNota[].
        Por ejemplo, un cuarto de tiempo son 1000/4 segundos, una octava parte son 1000/8 segundos, etc.*/
        tone(8, melodía[i], 1000/duraciónNota[i]);
        /*Como la función tone() no es bloqueante, el sketch sigue ejecutándose sin parar después de ella. Para
        evitar volver arriba del loop enseguida y distinguir las notas, establezco un tiempo mínimo entre ellas
        (la duración de la nota + 30% parece ir bien) parando el sketch */
        delay(1300/duraciónNota[i]);
        // Se deja de emitir la nota
        noTone(8);
    }
}
void loop(){}

```

En el sketch anterior hemos usado una serie de frecuencias concretas. ¿A qué notas musicales corresponden? ¿Cómo hemos sabido su valor? Bueno, lo primero que hemos de saber es que todo el conjunto de notas se dividen en "paquetes" que

EL MUNDO GENUINO-ARDUINO

se llaman "octavas", y cada octava tiene doce notas: do, do# (el símbolo "#" se lee "sostenido"), re, re#, mi, fa, fa#, sol, sol#, la, la# y si. Pero dentro del espectro audible hay una decena de octavas, por lo que en realidad hay varios dos, varios res, etc. Para distinguir las notas con el mismo nombre de diferentes octavas, tras su nombre se les añade un número indicando a qué octava pertenece; así podemos tener la nota mi4, la nota sol3, etc. Lo importante de todo esto es saber que una nota de una octava concreta (pongamos que el fa2) se corresponde con una onda de exactamente la mitad de frecuencia que la nota con el mismo nombre de la octava superior (en este caso, fa3).

Existe una fórmula matemática que permite obtener las frecuencias de todas las notas de todas las octavas. Dada la nota que queremos (la letra "n" de la fórmula, que ha de ser un entero entre 1 y 12: do=1, do#=2, re=3... hasta si=12) y dada la octava donde está (la letra "o" de la fórmula, que ha de ser un entero entre 1 y 10), se puede saber su frecuencia si calculamos la fórmula $440 \cdot e^{((o-3)+\frac{n-10}{12}) \cdot \ln(2)}$.

El siguiente sketch muestra las frecuencias correspondientes a 10 octavas; cada octava se mostrará por el "Serial monitor" separada por una línea de guiones. Las octavas que solemos escuchar en la mayoría de canciones son la 2^a, 3^a y 4^a.

Ejemplo 6.54

```
void setup(){
    int i,j;
    Serial.begin(9600);
    for(i=1; i<=10; i++){ //Recorro las escalas
        Serial.print("-----Escala ");
        Serial.println(i);
        for(j=1; j<=12; j++){ //Recorro las notas de esa escala
            //j=1 es un do, j=2 es un do#, j=3 es un re...
            Serial.print(j);
            Serial.print(" ");
            Serial.println(frecuencia(i,j));
        }
    }
}

void loop(){}
float frecuencia(float octava, float nota) {
    return (440.0*exp(((octava-3)+(nota-10)/12)*log(2)));
}
```

Otro código algo más sofisticado, que también reproduce una melodía, es el siguiente. Para ejecutarlo necesitamos el mismo circuito del ejemplo anterior:

Ejemplo 6.55

```

int longitud = 15; // Número de notas de la canción
/*El array notas[] tiene las notas de la canción
Se emplea la notación americana:
do=c, re=d, mi=e, fa=f, sol=g, la=a, si=b
Un espacio representa un silencio */
char notas[] = "ccggaagffeeddc ";
/*Duración de cada nota de la canción.
"2" dura el doble que "1", "4" dura el doble que "2", etc*/
int pulsacion[] = {1,1,1,1,1,2,1,1,1,1,1,2,4};
//Nombres de las notas de una escala
char nombres[] = { 'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C' };
//Sus correspondientes frecuencias
int tonos[] = { 523, 587, 659, 698, 783, 880, 98, 1047 };
int tempo = 350;
void setup() {
    pinMode(8, OUTPUT);
}
void loop() {
    int i;
    for (i = 0; i < longitud; i++) {
        if (notas[i] == ' ') {
            //Si hay silencio, se espera
            delay(pulsacion[i] * tempo);
        } else {
            //Si no, emite la nota
            playNota(notas[i]);
            //con la duración especificada
            delay(pulsacion[i]*tempo);
            noTone(8);
        }
    }
}
void playNota(char nota) {
    int i;
    /*Emite la frecuencia correspondiente al nombre de la nota. Para
    ello, busco en el array nombres[] si la nota especificada como parámetro está allí. */
    for (i = 0; i < 8; i++) {
        if (nombres[i] == nota) {
            tone(8,tonos[i]);
        }
    }
}

```

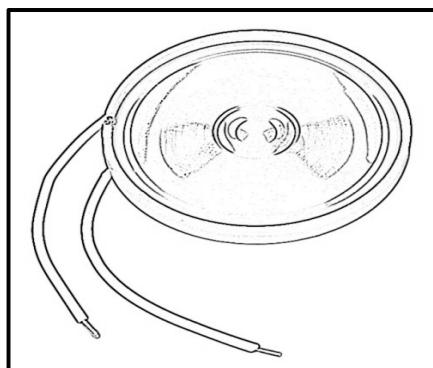
EL MUNDO GENUINO-ARDUINO

Y otro código de ejemplo más, muy parecido al anterior. En este caso, el mismo circuito reproducirá la nota musical especificada a través del canal serie.

Ejemplo 6.56

```
char nombres[] = {'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C'};  
int tonos[] = { 523, 587, 659, 698, 783, 880, 983, 1047 };  
char caracter;  
void setup() {  
    pinMode(8, OUTPUT);  
    Serial.begin(9600);  
}  
void loop() {  
    caracter = Serial.read();  
    if (caracter != -1) { //Si se ha escrito algo en el canal serie...  
        playNota(caracter); //...intento emitir la nota correspondiente  
    }  
}  
void playNota(char nota) {  
    int i;  
    for (i = 0; i < 8; i++) {  
        /*Si el carácter escrito en el "Serial monitor" coincide con alguno  
        correspondiente a alguna nota dentro del array, se reproduce. Si no, hay silencio */  
        if (nombres[i] == nota) {  
            tone(8,tonos[i]);  
            delay(500); //La emisión de la nota dura medio segundo  
            noTone(8);  
        }  
    }  
}
```

Uso de altavoces



En vez de zumbadores, podemos utilizar altavoces sencillos de poca potencia (0,5W, por ejemplo), como los productos 9151 o 10722 de Sparkfun. Al igual que los zumbadores, los altavoces generan ondas acústicas a partir de las señales eléctricas que reciben, pero el mecanismo físico para lograrlo es mucho más sofisticado, por lo que podremos lograr una calidad y un volumen de sonido bastante mayor.

CAPÍTULO 6: ENTRADAS Y SALIDAS

Para conectar los altavoces a nuestro circuito, debemos enchufar uno de sus dos terminales a tierra y el otro a un pin-hembra digital del Arduino configurado como salida, que será por donde enviaremos los pulsos de señal generados por la función *tone()*. Los altavoces suelen ser componentes polarizados, por lo que hay que fijarse si tiene alguna marca para indicar la polaridad de cada terminal (normalmente, el terminal positivo tiene un cable rojo y el negativo uno negro).

Es muy importante conectar un divisor de tensión en serie al altavoz para no dañar la placa. El valor mínimo de este divisor ha de ser de 100Ω , aunque si se utiliza un valor mayor, también estará bien. Hay que tener en cuenta, no obstante, que cuanta mayor resistencia tenga el divisor, menor será el volumen de sonido que puede generar el altavoz. En este sentido, es recomendable utilizar un potenciómetro (preferiblemente logarítmico) como divisor de tensión variable para así poder controlar el volumen del sonido convenientemente.

En configuraciones más sofisticadas a veces también se conecta un condensador "by-pass" (de $0,1\mu F$ está bien) entre los dos terminales del altavoz para protegerlo de posibles picos de señal provenientes del pin de salida digital de la placa cuando esta cambia su estado de HIGH a LOW, o viceversa.

Además, muchas veces el altavoz suele venir también acompañado de un condensador conectado en serie a él. La función de este condensador es actuar como filtro "pasa-altos". Un filtro "pasa-altos" conduce muy bien las señales eléctricas correspondientes a las frecuencias de sonido altas (concretamente, a las que son mayores que una determinada frecuencia llamada "frecuencia de corte", diferente según las características concretas de cada condensador) pero ofrece mucha resistencia a (y por tanto, elimina) las frecuencias de la señal que están por debajo de esa frecuencia de corte. Entre otras aplicaciones, los filtros pasa-altos se suelen utilizar mucho para hacer desaparecer la señal de frecuencia 0 (correspondiente a una señal eléctrica estable y continua que siempre aparece como "colchón" permanente bajo el resto de señales que forman realmente el sonido), ya que tan solo sirve para calentar los altavoces y malgastar energía inútilmente consumiendo más rápido la carga de las pilas/baterías (si es el caso).

En otro orden de cosas, es importante conocer las dos características técnicas más relevantes de un altavoz: su resistencia y su potencia de trabajo. La resistencia del altavoz se suele llamar "impedancia" (para señalar que, en realidad, esta característica no tiene un valor constante sino que depende de la frecuencia de la señal, pero en esto no profundizaremos). En la mayoría de proyectos donde interviene una placa Arduino los altavoces más habitualmente utilizados son los de 4Ω o 8Ω nominales.

EL MUNDO GENUINO-ARDUINO

La potencia de trabajo de un altavoz es la cantidad de energía que puede consumir en un segundo sin problemas (recordemos que está definida como $P=V \cdot I = I^2 \cdot R = V^2/R$), más allá de la cual podría dañarse. Este dato nos sirve para conocer la "cantidad de volumen" máximo de sonido que puede proporcionar ese altavoz, ya que cuanto mayor sea su potencia de trabajo, mayor volumen será capaz de emitir.

Si conectáramos directamente (sin divisor de tensión) un altavoz de 8Ω a un pin de salida de nuestra placa Arduino (el cual ya sabemos que aporta 5V en estado HIGH), la potencia consumida (y por tanto, el volumen emitido) sería la máxima posible: $P=(5V)^2/8\Omega=3,125W$. No obstante, suponiendo que el altavoz soporte esta potencia, tendríamos un problema: la intensidad que fluiría por el pin de salida sería $I=V/R=5V/8\Omega=625mA$ (también podríamos haber llegado al mismo resultado mediante: $I=(P/R)^{1/2}=(3,125W/8\Omega)^{1/2}=625mA$), cantidad que es muy superior al límite máximo de intensidad soportado por estos pines, el cual ronda los 40/50mA. Por tanto, para no freír estos pines es necesario incluir un divisor de tensión que reduzca la intensidad y potencia generadas a los niveles aceptados por la placa. De ahí lo comentado en párrafos anteriores sobre la necesidad de conectar una resistencia en serie al altavoz.

Sin embargo, tampoco es bueno pasarse en el valor del divisor de tensión: si conectáramos por ejemplo un resistor de $4,7K\Omega$ entre un pin de salida de la placa Arduino y ese mismo altavoz, la tensión a la que estaría sometido dicho altavoz sería $V=8\Omega \cdot 5V/(8\Omega+4,7K\Omega)= 8,5mV$, y por tanto tan solo recibiría una intensidad de $I=V/R= 8,5mV/8\Omega \approx 1mA$. Esto implicaría que la potencia recibida sería de 0,008 milivatios ($P=V \cdot I=0,0085V \cdot 0,001^a = 0,0000085W$). Es decir, un sonido inaudible. Si probamos en cambio con un valor para el resistor de por ejemplo 120Ω , la tensión a la que se sometería entonces el altavoz sería de $V=8\Omega \cdot 5V/(8\Omega+120\Omega)= 312,5mV$ y la intensidad recibida sería de $I=V/R=312,5mV/8\Omega \approx 39mA$, con lo que la potencia resultante sería de $P=V \cdot I=0,3125V \cdot 0,039^a = 12.2mW$. En este caso vemos que aun aportando por el pin de salida el máximo de corriente admitida, la potencia generada (aunque mucho mayor que en el caso del divisor de $4,7K\Omega$) sigue siendo no demasiado elevada. Esto nos lleva a una conclusión: si deseamos más volumen, deberemos utilizar un amplificador.

Amplificación simple del sonido

Un amplificador sirve para aportar a un altavoz una determinada potencia, mayor de la que aporta la fuente del sonido (en nuestro caso, la placa Arduino) por sí misma. A más potencia recibida, el altavoz podrá vibrar con más fuerza y emitir el sonido a un volumen mayor. Para conseguir su objetivo, los amplificadores pueden

"jugar" con aumentar o bien el voltaje, o bien la intensidad aportados al altavoz, o bien una combinación de ambos (esto es fácil verlo si recordamos que $P=V \cdot I$).

Lo más sencillo en la práctica es utilizar amplificadores en forma de placas breakout, ya que nos facilitan las conexiones y además incorporan circuitería suplementaria que estabiliza las señales. La idea es, en general, por un lado conectar el pin digital de salida de audio de la placa Arduino a la entrada adecuada de la plaquita-amplificador, por otro lado conectar la salida de la plaquita-amplificador a la entrada de señal del altavoz, y compartir una tierra común con los tres dispositivos implicados: Arduino, plaquita-amplificador y altavoz.

Una placa breakout que incluye un amplificador de audio (concretamente, el TPA2005D1) es el producto nº 11044 de Sparkfun. Esta placa, alimentada con 5V, puede proporcionar a un altavoz de 8Ω una potencia de hasta 1,4W con una distorsión máxima (lo que técnicamente se llama "THD + N" (es decir, Total Harmonic Distortion + Noise) del 0,2%. Tiene un conector etiquetado como "IN+", donde deberemos enchufar la salida digital de la placa Arduino por donde emite `tone()` y un conector "IN-", que ha de ir a tierra; por otro lado tiene dos conectores más ("OUT+" y "OUT-") para el altavoz de 8Ω . Finalmente, tiene un conector para su alimentación, otro para tierra y otro etiquetado como SDN, el cual, si se conecta a tierra o a una señal LOW apagará el amplificador (para ahorrar consumo). También admite la posibilidad de soldar un potenciómetro de $10K\Omega$ (aunque no lo incorpora de serie), para ajustar el nivel de amplificación del sonido.

Otra placa muy parecida es la llamada "LM4889 Audio amplifier", distribuida por Modern Devices. Esta placa, alimentada con 5V, puede proporcionar a un altavoz de 8Ω una potencia de hasta 1W con una distorsión máxima del 0,2%. Esta placa tiene un solo conector para la entrada de sonido proveniente de la placa Arduino, dos conectores (+ y -) para el altavoz de 8Ω , un conector para la alimentación, otro para tierra y otro etiquetado como SHD, el cual, si se conecta a tierra o a una señal LOW apaga el amplificador (para ahorrar consumo). También dispone de un potenciómetro ya preensamblado para poder alterar la potencia proporcionada y así regular el volumen de salida.

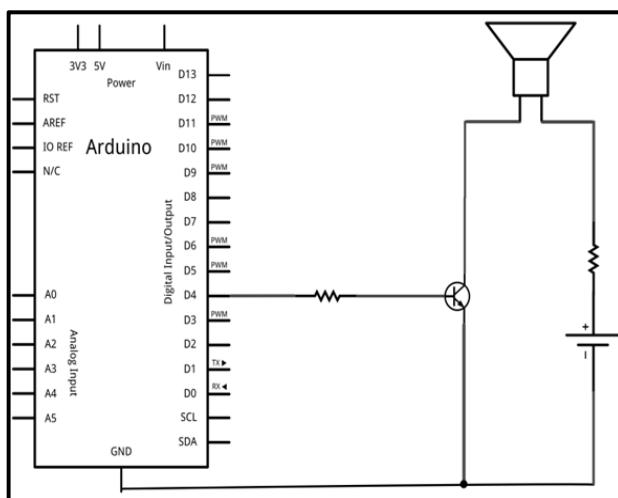
De todas formas, podemos aumentar el volumen del sonido emitido por nuestra placa Arduino sin necesidad de utilizar ningún amplificador específico como los descritos en párrafos anteriores. Estos dispositivos están especializados en reproducir ondas acústicas complejas sin alterar el sonido original y mantener la proporción de frecuencias intacta, pero en realidad, la placa Arduino no es capaz de "generar" audio real: la función `tone()` tan solo emite una onda cuadrada que oscila

EL MUNDO GENUINO-ARDUINO

entre 5V y 0V a una determinada frecuencia única. Esta gran simplicidad de la onda permite que para amplificar esta señal (es decir, para aumentar la potencia enviada al altavoz) nos baste con un simple transistor bipolar y nada más.

La idea es simplemente aumentar la intensidad de corriente que circula por el altavoz sin aumentar la que sale del pin de la placa Arduino por donde se emite la señal a escuchar (la cual está limitada, recordemos, a tan solo 40/50mA). En el esquema siguiente se muestra un circuito donde se implementa esta idea. Tal como se puede ver, se utiliza un transistor NPN (como por ejemplo el 2N4401 u otro similar) como amplificador. Su funcionamiento es muy parecido al que vimos cuando estudiamos el control de motores DC: la única diferencia destacable es que ahora la base del transistor está conectada a un pin de salida digital (en vez de a una salida PWM), porque allí será por donde se emitirán los tonos musicales mediante el uso de `tone()`, función cuyo comportamiento, recordemos, es totalmente digital. La resistencia conectada a la base del transistor si es de 100Ω ya es suficiente.

La fuente de alimentación externa puede ser de 5V o más, siempre que la potencia aportada al altavoz no exceda la máxima admitida. Es fácil calcular, usando la Ley de Ohm, cuánta potencia aporta si conocemos la tensión generada por la fuente y el valor del divisor de tensión del altavoz (es decir, de la resistencia conectada en serie a él). De todas formas, aunque se elija una fuente de 5V, es muy recomendable no utilizar la propia placa Arduino como fuente para evitar posibles ruidos e inestabilidades en la señal.



Sonidos pregrabados

Añadir audio de calidad a un proyecto con Arduino es difícil: el microcontrolador que incorpora no es lo suficientemente capaz para manejar la cantidad de datos necesaria para poder reproducir sonidos de calidad. No obstante, existen algunos proyectos que intentan superar estas limitaciones.

La librería "SimpleSDAudio"

Uno de ellos es la librería "TMRh20", descargable desde (<https://github.com/TMRh20/TMRpcm/wiki>) y otro es la librería "SimpleSDAudio", descargable desde <http://hackerspace-ffm.de/wiki/index.php?title=SimpleSDAudio>; ambos permiten reproducir ficheros de sonidos almacenados en una tarjeta SD con una calidad decente con un mínimo de hardware adicional. Concretamente, la librería "SimpleSDAudio" es capaz de reproducir audio con una resolución de 8 bits y una frecuencia de muestreo de 62,5KHz ("fullrate") o bien de 31,25KHz ("halfrate"), en modo mono o modo estéreo. Eso sí, requiere 1,3 kilobytes de RAM para poder funcionar.

Breve nota sobre las características de un fichero de audio

No todos los ficheros de audio son iguales: unos reproducen el sonido con más calidad que otros. Pero ¿qué significa exactamente que un sonido tenga "más calidad"? Cuando se captura un sonido (por medio de un micrófono o similar) se produce un proceso de conversión de la onda sonora (analógica) a pulsos eléctricos digitales (bits). La frecuencia a la que se toman muestras de esa onda sonora para guardarlas en un fichero es la llamada "frecuencia de muestreo". Es decir, este parámetro indica el número de veces por segundo que el micrófono ha tomado información del sonido. Es evidente que cuanto mayor sea la frecuencia de muestreo, la captura de la señal será más fidedigna al sonido real porque dejará menos intervalos sin medir. No confundir frecuencia de muestreo con la frecuencia del sonido. Los valores más usuales empleados en las grabaciones digitales son 11,025Hz para grabaciones de voz, 22,050Hz para grabaciones de música con calidad mediana y 44,100Hz para grabaciones de música con alta calidad.

Otro dato importante es la resolución. Representa la cantidad de información que se captura en cada muestra del sonido. De nada sirve tener una gran frecuencia de muestreo si luego en cada muestra no se ha capturado cómo era el sonido en ese momento de la forma más precisa posible. Los valores más comunes para la resolución son 8 bits por lectura (por lo que solo se permiten 256 valores posibles diferentes para almacenar el estado del sonido en ese momento) y 16 bits por

EL MUNDO GENUINO-ARDUINO

lectura (por lo que ofrece una escala de 65,536 y por lo tanto, permite capturas mucho más completas y precisas). Además, tenemos otro dato a considerar, que es el número de canales: si se realiza un solo registro sonoro estamos hablando de una señal monofónica y se realizan dos registros simultáneos es una señal estereofónica.

Por otro lado, una vez grabado el audio en un fichero, podemos tener un problema: que ese fichero ocupe mucho espacio en disco. Es fácil comprobar que si multiplicamos el número de muestras por segundo (la "frecuencia de muestreo") por el número de segundos que dura la grabación por los bits almacenados en cada muestra por el número de canales usados, obtenemos una cantidad de bits muy elevada. Por tanto, en la mayoría de ocasiones, estos ficheros se guardan de forma comprimida.

Los códecs de audio son algoritmos matemáticos que permiten comprimir los datos, haciendo que ocupen mucho menos espacio. Hay códecs que comprimen más o menos que otros, a cambio de perder más o menos calidad de sonido. El problema es que el aparato reproductor de ese audio ha de poder entender el códec particular con el que se comprimió un determinado fichero, y si no es así, no lo podrá reproducir. Los códecs de audio más usuales son el conocido popularmente como "MP3" (blindado por un complejo sistema de patentes) o el "Vorbis" (basado en estándares abiertos y libres), entre muchísimos más. Si el fichero no está comprimido por ningún códec, se suele decir que está en formato PCM ("Pulse Code Modulation").

Normalmente, la extensión de un fichero de audio indica el códec que se ha utilizado en él (.mp3 para un fichero codificado en MP3, etc.), pero hay casos en que no es tan fácil. Por ejemplo, la extensión ".wav" es la que se acostumbra a emplear en Windows para identificar los ficheros de audio digital, pero los datos de los ficheros ".wav" pueden estar en formato PCM (sin comprimir) o pueden haber sido comprimidos con cualquiera de los códecs disponibles para Windows.

Los ficheros deben estar grabados en una tarjeta SD o SDHC porque la memoria EEPROM de la placa Arduino es muy limitada y es muy difícil poder guardar allí música de cierta duración. Deberemos, por tanto, utilizar un shield o módulo que incorpore un zócalo SD/microSD (o conectar directamente la tarjeta SD a nuestro circuito mediante una ristra de pines soldados a sus terminales).

La librería "SimpleSDAudio" asume que la tarjeta SD se comunica vía SPI con la placa Arduino, por lo que por defecto utiliza para ello los pines estándares 11, 12 y

13, y el nº 4 como canal CS. Si la tarjeta que tengamos conectada utilizara otro pin como canal CS, este se debería especificar entonces en el código fuente de nuestro sketch. Además, requiere obligatoriamente el uso de los pines PWM 9 y 10 de la placa Arduino UNO para la salida del audio (si la salida es mono tan solo se requiere el pin 9). Por tanto, suponiendo que queremos conectar un altavoz mono a nuestra placa Arduino, la manera más sencilla sería enchufarlo por un lado al pin 9 a través de una resistencia en serie de 100Ω y por otro a tierra. Otros ejemplos de conexión se muestran dentro del fichero "SimpleSDAudio.h", incluido dentro de la librería.

El "Wave Shield" de Adafruit

La gente de Adafruit ha diseñado un shield (el "Wave Shield") que permite reproducir ficheros de audio (de cualquier duración) que tengan las siguientes características: que sean "mono", que tengan una frecuencia de muestreo de 22KHz, una resolución de 12 bits y que no tengan compresión.

Los ficheros deberán estar grabados en una tarjeta SD (o SDHC) formateada en FAT16 o FAT32, la cual debemos tener introducida en el zócalo correspondiente del shield. Estos ficheros solo pueden estar almacenados en la carpeta "raíz", y solo puede ser reproducido uno en cada momento.

El shield dispone de un zócalo "jack" de 3,5mm estándar para poderle conectar unos auriculares, y también de dos conectores para enchufar los cables de un altavoz (los cuales se activan cuando detectan que no hay auriculares). Estos altavoces pueden ser de 8 ohmios o de 4 ohmios; en el primer caso recibirán una potencia de $1/8W$ y en el segundo una de $\frac{1}{4} W$ gracias a un amplificador integrado en el shield, el TS922IN. En cualquier caso, el volumen de salida se puede controlar con un potenciómetro logarítmico también incorporado en el shield.

Los pines 10, 11, 12 y 13 del shield son usados por la tarjeta SD para comunicarla vía SPI con la placa Arduino. Los pines 2, 3, 4 y 5 del shield son usados por defecto por el conversor digital-analógico para comunicarse con la placa Arduino. Así pues, solo quedan disponibles los pines digitales 6, 7, 8 y 9, además de los pines de entrada analógicos (que pueden actuar también como pines de entrada/salida digitales extra).

Los ficheros han de estar en el formato mencionado anteriormente (22KHz, 12 bits mono y sin comprimir) para ser reproducidos por el "Wave Shield" ya que el tratamiento de ficheros comprimidos requiere un chip especializado de potencia y precio elevado, y por eso no está incluido en este shield. Para saber si un fichero de audio tiene el formato requerido, podemos consultar la información mostrada en el

EL MUNDO GENUINO-ARDUINO

cuadro que aparece en cualquier sistema operativo cuando hacemos clic con el botón derecho sobre el icono de ese fichero y seleccionamos la opción "Propiedades". En caso de que el formato actual no sea el adecuado, una manera de convertirlo al que nos interesa es utilizando un editor de audio cualquiera, como por ejemplo Audacity (<http://audacity.sourceforge.net>), el cual es libre y multiplataforma.

Si usamos Audacity, el proceso es el siguiente: tras abrir el fichero, para convertir el fichero en mono (si no lo es ya) debemos seleccionar las dos pistas estéreo e ir al menú "Tracks > Stereo Track to Mono". Para pasarlo a una resolución de 12 bits (en realidad 16, pero también funciona) debemos clicar en el título de la pista y seleccionar del cuadro desplegable la opción llamada "Set Sample Format -> 16-bit". Para pasarlo a una frecuencia de 22KHz (o menos), debemos seleccionar del mismo cuadro desplegable anterior la opción "Set rate -> 22.050Hz". Para guardar el fichero en formato no comprimido, debemos usar la opción "Export as WAV" del menú "File" y elegir el formato "Other uncompressed files"; seguidamente hay que clicar en el botón "Options" y seleccionar el header "WAV (Microsoft)" y el encoding "Signed 16 bit PCM". Una vez hecho todo esto, podremos clicar finalmente en el botón "Save".

El "Wave Shield" controla mediante una librería específica desarrollada por la gente de Adafruit y descargable de <http://code.google.com/p/wavehc>. Esta librería nos permite reproducir sonidos cuando se pulsa un botón, cuando un sensor se activa, cuando un dato se recibe a través del canal serie, etc. El sonido se reproduce de forma asíncrona, por lo que la placa Arduino puede seguir trabajando mientras el sonido se está reproduciendo. Por razones de espacio no podemos profundizar en su estudio, pero afortunadamente está ampliamente documentada y proporciona muchos ejemplos. Concretamente, el sketch de ejemplo "dap_hc.pde" es el que permite reproducir todos los ficheros grabados en la tarjeta SD uno tras otro sin fin; recomiendo su estudio pormenorizado para comprender las interioridades de esta librería. Otros códigos de ejemplo interesantes y muy bien comentados se pueden encontrar en <https://learn.adafruit.com/adafruit-wave-shield-audio-shield-for-arduino/examples>.

Shields que reproducen MP3

Si lo que deseamos es reproducir ficheros MP3 (a 192Kbps) tal como lo solemos hacer en nuestro reproductor de música portátil o desde nuestro computador, existen shields adecuados para eso. Por ejemplo, el "**MP3 Player Shield**" de Sparkfun. Al igual que el "Wave Shield" de Adafruit, incluye un zócalo para insertar una tarjeta SD con los ficheros de audio, y ofrece una salida jack de 3,5mm estéreo para los auriculares además de dos conexiones ("R/L" y "-") para enchufar un altavoz

pequeño (o un amplificador externo). Este shield utiliza el chip decodificador de MP3 llamado VS1053B, el cual también es capaz de reproducir audio codificado en los códigos Vorbis, AAC y WMA, además de poder interpretar ficheros MIDI. Para trabajar con él en la página de Sparkfun proponen varios sketches de ejemplo, pero son de un nivel relativamente complicado; afortunadamente, existe una librería llamada "SFEMP3Shield" que facilita mucho su programación, descargable desde <https://github.com/madsci1016/Sparkfun-MP3-Player-Shield-Arduino-Library>.

Otro shield que puede reproducir ficheros MP3 (y Vorbis) es el "**Music Shield**" de Seeedstudio, también basado en el chip VS1053B. Se programa mediante una librería propia llamada "Music", descargable de la página del producto. Esta librería se basa en una librería independiente –pero incluida en la descarga– para acceder a los ficheros de la tarjeta SD, llamada "Fat16" (<https://github.com/greiman/Fat16>). La librería "Fat16" es más rápida y ligera que la librería oficial de Arduino, pero solo puede trabajar con el formato FAT16, por lo que la tarjeta SD que se use con este shield solo puede estar formateada en FAT16. Como novedad, este shield incorpora un conector jack "LINE IN" que permite la grabación de sonido, aunque esta funcionalidad solamente vale cuando es acoplada a una placa Arduino Mega. Recomiendo consultar su documentación online, que es muy completa y clara.

Un shield algo diferente de los anteriores es el "**Music Instrument Shield**" de Sparkfun (producto nº **10587**). Aunque incorpora el mismo chip VS1053B que el que viene en el "MP3 Player Shield", está preconfigurado para solo actuar como reproductor MIDI. Esto significa que mediante el envío de determinados comandos a través del canal serie, este chip es capaz de reproducir una gran cantidad de sonidos (pianos, vientos, percusiones, efectos especiales, etc.) que tiene pregrabados de fábrica. Incluso pueden sonar hasta 31 instrumentos diferentes a la vez. El shield ofrece un conector jack de 1/8" para conectar un altavoz o unos auriculares. Recomiendo consultar la página <http://www.sparkfun.com/tutorials/302> para comprender en profundidad el uso de este shield, además de leer y probar los muy ilustrativos códigos Arduino de ejemplo disponibles en la página del producto.

Módulos de audio

Si, en vez de shields, lo que queremos es utilizar módulos independientes capaces de almacenar ficheros de audio y de reproducirlos, tenemos unas cuantas alternativas. Por ejemplo, Sparkfun ofrece una placa breakout para el mismo chip VS1053B que viene en su "MP3 Player Shield", a la cual se pueden conectar directamente las salidas de audio. En la página del producto (con código **9943**) se ofrecen códigos Arduino de ejemplo de manejo de esta placa, pero son de un nivel más avanzado que el de este libro, así que recomendamos otras alternativas.

EL MUNDO GENUINO-ARDUINO

DFRobot fabrica la plaquita "**DFRduino Player**". Tiene un zócalo para alojar una tarjeta SD formateada en FAT16 donde se han de almacenar los ficheros de audio y se basa en el chip decodificador de MP3 VS1003 (pudiendo reproducir ficheros en formato WAV, MP3 y MIDI). Se puede comunicar con nuestra placa Arduino por el canal serie o bien vía I²C (seleccionable mediante un jumper). En el caso de utilizar la comunicación I²C, además del cable de alimentación (5V) y tierra, se ha de usar un cable conectando el pin "DO" del módulo y el pin SDA de la placa Arduino, y otro conectando el pin "DI" del módulo y el pin SCL de la placa Arduino. En el caso de utilizar la comunicación serie, además del cable de alimentación (5V) y tierra, se ha de usar un cable conectando el pin "DO" del módulo y el pin RX de la placa Arduino, y otro conectando el pin "DI" del módulo y el pin TX de la placa Arduino. Para conectar los altavoces (se pueden acoplar hasta dos a la vez), este módulo ofrece dos parejas de terminales + y – que aportan una potencia de hasta 3W por altavoz.

Esta placa reconoce diferentes comandos, tales como reproducir la siguiente o anterior canción, pausar o continuar la reproducción o cambiar el volumen. Dependiendo del tipo de conexión (serie o I²C), estos comandos son diferentes y su envío se debe escribir en nuestro sketch de diferente manera. Recomiendo en este sentido consultar su documentación online para conocer los detalles. Lo que sí que hay que tener presente es que esta placa solo funcionará si existe una carpeta llamada "sound" en la raíz de la tarjeta SD, y si dentro de ella los ficheros existentes tienen las extensiones "wma", "wav", "mid" o "mp3".

Otro módulo con zócalo microSD (el cual solo admite tarjetas formateadas en FAT16) y conectores para altavoces es el "**SOMO-14D**" de 4DSystems. Esta placa tiene la particularidad de que solo es capaz de reproducir ficheros en formato ADPCM (.ad4) a 32KHz y 4bits (los cuales, además, han de estar obligatoriamente grabados directamente en la raíz de la tarjeta). Para convertir nuestros ficheros WAV o MP3 en este formato, desde la página web del producto podemos descargarnos gratuitamente un programa conversor (para Windows tan solo, no obstante).

Lo interesante de este módulo es que puede funcionar en dos modos. En el "serial mode", operaciones como "reproducir", "pausar", "parar" o "cambiar volumen" pueden ser ordenadas por la placa Arduino a través de comandos hexadecimales de 16 bits enviados por el canal serie. En el "key mode", la placa puede funcionar de forma autónoma sin necesidad de ninguna placa Arduino que la controle: tan solo se requiere un circuito formado por tres pulsadores, una batería de 3V y un altavoz convenientemente conectados (el circuito concreto se muestra en el datasheet del producto). Si usamos el "serial mode", las conexiones son:

Placa breakout	Exterior
Nº 1	-
Nº 2	-
Nº 3 (CLK)	Una salida digital de Arduino
Nº 4 (DATA)	Una salida digital de Arduino
Nº 5	*
Nº 6	-
Nº 7	-
Nº 8	Pin 3V3 de Arduino
Nº 9	Pin GND de Arduino
Nº 10 (RESET)	Una salida digital de Arduino
Nº 11	Terminal de altavoz (8Ω,1W)
Nº 12	Terminal de altavoz (8Ω,1W)
Nº 13	-
Nº 14	-

Todos los pines del módulo trabajan a 3V por lo que para no dañarlo necesitamos colocar resistencias de 470Ω en serie en los conectores nº 3, 4, 5 y 10 de manera que el nivel de 5V de las salidas de la placa Arduino se adapte convenientemente.

El pin nº 5 de la placa breakout se puede conectar opcionalmente a un pin digital de la placa Arduino configurado como entrada, y a la vez, al ánodo de un LED (cuyo cátodo se conectará a tierra a través de un divisor de tensión). Este pin emitirá una señal HIGH mientras se esté reproduciendo algún fichero de audio, por lo que si eso ocurre, el LED se encenderá y la placa Arduino detectará ese valor para poderlo tener en cuenta.

Para controlar este módulo no disponemos de ninguna librería específica, pero podemos utilizar como referencia los códigos de ejemplo disponibles en <http://bit.ly/bricotuto-somo14d>, los cuales aportan diferentes funciones Arduino para las acciones más habituales, como reproducir una canción concreta (`playSong(nº cancion);`), pausarla (`pausePlay();`), reproducir la siguiente canción (`nextPlay();`), incrementar o disminuir el volumen (`incVol();` y `decVol();`), etc. En cualquier caso (también en el "key mode"), se recomienda la ayuda de un amplificador externo.

Existe un módulo muy parecido al "SOMO-14D" comercializado por Sparkfun con el código de producto **nº 11125**. Como novedad incluye un conector JST para enchufar una batería LiPo, pero por lo demás es funcionalmente idéntico.

EL MUNDO GENUINO-ARDUINO

Otro módulo más es el llamado "**SmartWAV**" de Vizictechnologies. Este es el módulo que ofrece más calidad de sonido de los hasta ahora nombrados. Concretamente, es capaz de emitir en estéreo, a 16 bits de resolución y a 48KHz de frecuencia de muestreo (es decir, calidad CD). Además, incorpora un potenciómetro digital (el chip AD5206) que permite el control del volumen en 255 pasos y la reproducción a distintas velocidades. Por otro lado, es capaz de reconocer tarjetas tanto microSD como microSDHC, formateadas tanto en FAT16 como en FAT32, por lo que los nombres de los ficheros pueden ser más largos de ocho caracteres; y también tiene capacidad para gestionar varios niveles de carpetas. Respecto a las conexiones de audio, dispone de un conector jack de 3,5mm para enchufar unos auriculares o un altavoz o un amplificador externo.

Igual que el "SOMO-14D", el "SmartWAV" tiene dos modos de funcionamiento: "modo serie" y "modo autónomo". En el modo serie operaciones como "reproducir", "pausar", "parar" o "cambiar volumen" pueden ser ejecutadas por la placa Arduino mediante funciones muy sencillas (del tipo *objetoSwav.playTrack()*/*objetoSwav.stopTrack()*) pertenecientes a una librería propia llamada "SmartWAV". En realidad, esta librería (descargable de la web del producto) lo que hace esocultar los detalles internos de la comunicación entre placa Arduino y módulo, la cual se establece a través del canal serie.

En el modo autónomo, el módulo puede funcionar sin necesidad de ninguna placa Arduino que la controle: tan solo se requiere un circuito formado por cinco pulsadores (para las señales de Reproducir/Pausa, Siguiente, Rebobinado, Volumen+ y Volumen-) y una batería de 3V convenientemente conectados (el circuito concreto se muestra en la web del producto). Para que la placa funcione en modo autónomo, es necesario que su pin "MODE" esté conectado a tierra: si no lo está estaremos usando el "modo serie", en cuyo caso las conexiones necesarias son las siguientes:

Placa breakout	Exterior
GND (superior izquierda)	Pin GND de Arduino
3V3 (superior izquierda)	Pin 3V3 de Arduino
TX	Pin RX de Arduino
RX	Pin TX de Arduino
RST	Pin RESET de Arduino
A	*

El pin A del módulo tiene idéntica función y conexiones que el pin nº 5 del módulo "SOMO-14D", explicado en párrafos anteriores. Los pines GND y 3V3 a la derecha de la placa están reservados para la comunicación FTDI.

Otro módulo digno de mención es el "**Embedded MP3 Module**" de OpenElectronics, que puede ser controlado mediante el envío de determinados comandos propios a través del canal serie (aunque también puede funcionar en modo autónomo sin necesidad de microcontrolador si diseñamos el circuito adecuado). Los esquemas de conexiones y la lista concreta de comandos se pueden encontrar en <http://www.open-electronics.org/embedded-mp3-module>.

Finalmente, otro módulo a destacar es el "MP3 Trigger", distribuido por Sparkfun con el código de producto **11029**. Esta plaquita almacena en una tarjeta SD (formateada en FAT16 o FAT32) una serie de ficheros MP3 (a 192Kbps) que se han de llamar obligatoriamente de la forma TRACK001.mp3, TRACK002.mp3 hasta TRACK256.mp3 como máximo. Dispone de una salida de audio en forma de jack de 3,5mm para poder conectar altavoces o auriculares o un amplificador externo. Para controlarla desde Arduino, su pin "USBVCC" se ha de conectar al pin de 5V de Arduino (para recibir la alimentación), su pin "GND" se ha de conectar al pin GND de Arduino (para conectar a tierra), su pin "RX" se ha de conectar al pin TX de Arduino y su pin "TX" al pin RX de Arduino. Puede ser controlado mediante el envío de determinados comandos propios a través del canal serie consultables en el datasheet del producto, pero afortunadamente existe una librería (descargable en <https://github.com/sansumbrella/MP3Trigger-for-Arduino>), que nos facilita mucho las cosas, ya que se encarga de gestionar tanto la comunicación serie entre plaquita y Arduino como de las funciones de reproducción de los ficheros MP3 (las cuales son muy sencillas, de tipo *objetoTrigger.setVolume()* o *objetoTrigger.stop()* por ejemplo).

Lo interesante del "MP3 Trigger" es que puede ser utilizado sin intermediación de ninguna placa Arduino gracias a que dispone de hasta 18 parejas de conectores (uno para la señal de entrada y otro para tierra) que permiten poderle acoplar cualquier tipo de sensor o pulsador, de tal forma que al recibir una determinada señal de ellos se reproduzca automáticamente un determinado fichero MP3. Para aprender a utilizar este modo de funcionamiento, remito a la documentación disponible en la página web del producto en Sparkfun.

Reproductores de voz

Sparkfun distribuye como producto nº **10661** el llamado "VoiceBox Shield", el cual incorpora el chip SpeakJet de Magnevation. Este chip es un sintetizador de sonidos y de voz, y el "VoiceBox Shield" lo utiliza para enviarle a través del canal serie una serie de comandos propios que acaban transformándose en una nítida voz robótica mediante la conexión de un altavoz a alguno de sus pines de salida. El vocabulario es infinito porque funciona a partir de fonemas y sonidos sintetizados, y la combinación concreta de comandos hace que se varíe el tono, la velocidad, el

EL MUNDO GENUINO-ARDUINO

volumen del audio generado, etc. Además del chip SpeakJet, este shield también incorpora un chip amplificador de audio interno junto con un potenciómetro (manipulable mediante un pequeño destornillador) para regular el volumen, un conector jack de audio estándar de 3,5mm para conectar un altavoz y también un par de conectores etiquetados como SPK+ y SPK- para enchufar asimismo un altavoz o un amplificador externo.

El envío por parte de la placa Arduino de los sonidos que ha de sintetizar el shield se realiza por el canal serie, pero debido al diseño del shield, este envío no se puede realizar por el pin TX hardware de la placa Arduino, sino que ha de transmitirse por su pin digital nº 2. Por tanto, se debe utilizar la librería SoftwareSerial para comunicarse con el shield. Una ventaja de esto es que los pines serie por hardware (nº 0 y nº 1) quedan libres para otro uso. Probemos su funcionamiento con un ejemplo simple, donde se reproduce la palabra "hello":

Ejemplo 6.57

```
#include <SoftwareSerial.h>
/* El pin 3 envía los datos al módulo. El pin 2 no se usa (no se reciben datos). */
SoftwareSerial miserie(2,3);
void setup(){
    miserie.begin(9600);
/*El array "frase" contiene un conjunto de códigos numéricos correspondientes a fonemas (ingleses)
que tiene pregrabados el chip. Colocándolos unos tras otros podemos formar las palabras y frases que
deseemos. Concretamente, el valor del array en este ejemplo hace que el shield reproduzca (a través de
un altavoz) la palabra "hello". También se han introducido códigos para definir la velocidad o el tono.
Para conocer los distintos fonemas asociados a los códigos numéricos, se recomienda consultar el
datasheet.*/
    char frase[] = {20,96,21,114,22,88,23,5,183,7,159,146,164,0};
/*El valor final del array siempre ha de ser 0 para indicar que se ha llegado al final de la palabra o frase
a pronunciar.*/
    miserie.print(frase);
}
void loop(){}
}
```

Concretamente, para decir la palabra "hello" aproximadamente, los fonemas necesarios son el código 183 (sonido "h"), el 159 (sonido "eh"), el 146 (sonido "lu") y el 164 (sonido "oh"). Otros códigos son el 20,96 (para establecer el volumen hasta nueva orden; el volumen mínimo es 20,0 y el máximo es 20,127); el 21,114 (para establecer la velocidad); el 22,88 (para establecer el tono de voz a 88Hz); el 23,5 (para establecer el timbre de la voz; para un sonido profundo sería 23,0 y para un sonido agudo y metálico sería 23,15); el 7 (para reproducir el siguiente fonema el doble de

rápido que usualmente), el 4 (para añadir una breve pausa entre fonemas), etc. Conjugando correctamente los valores de tono y velocidad, incluso podríamos hacer que el chip cantara.

A partir de aquí, no debería ser difícil escribir un sketch Arduino que, dependiendo de la activación de pulsadores u otro tipo de entradas provenientes de diferentes sensores, emitiera una frase u otra. Existe un software (solo para sistemas Windows) desarrollado por la misma empresa fabricante del chip Speakjet llamado Phrase-A-Lator que contiene un extenso diccionario de transcripciones fonéticas de palabras del idioma inglés, listas para ser usadas en nuestro código Arduino. Pero lo más interesante es que permite obtener los códigos equivalentes a las palabras deseadas de una forma muy intuitiva.

Otra alternativa por si no queremos aprender los códigos numéricos del chip Speakjet es utilizar en conjunción con este el chip TTS256. Este chip actúa como un intermediario entre el usuario y el chip Speakjet, ya que permite recibir a través del canal serie una cadena de texto literal (en inglés, eso sí) para convertirla en los códigos adecuados y entonces pasárselos al chip Speakjet. De hecho, el "**SpeakJet Shield TTS**" de <http://www.droidbuilder.com> es un shield bastante similar al VoiceBox de Sparkfun pero con la ventaja de que incorpora ambos chips, por lo que su programación es realmente sencilla ya que tan solo es necesario el uso de un objeto SoftwareSerial y su función *print()*. Desgraciadamente, solo se distribuye en forma de kit, por lo que es necesario soldar siguiendo las instrucciones indicadas en su página.

Una competencia directa al "VoiceBox Shield" de Sparkfun es el "**GinSing Shield**" de GinSingSound. Este shield (que se puede adquirir completamente ensamblado o en forma de kit) viene con el chip sintetizador Babblebot, el chip amplificador NJM386 y (al igual que el producto de Sparkfun) dispone de un conector de salida de audio de tipo jack de 3,5mm para poder enchufar unos auriculares.

Ambos shields son bastante parecidos, aunque el chip SpeakJet está más enfocado a facilitar la síntesis del habla y el chip BabbleBot es más bien un generador de sonidos complejos, que permite la creación de efectos, la reproducción de hasta seis canales de audio a la vez, la síntesis de música y voz, etc. De hecho, el chip BabbleBot que viene de fábrica en el shield GinSing está preconfigurado para trabajar en cada momento en un modo de funcionamiento de entre cuatro modos posibles: activación de efectos sonoros predefinidos o creados por uno mismo ("preset mode"), creación de un instrumento musical polifónico ("poly mode"), generación del habla ("voice mode") y ejecución de síntesis de una onda de audio ("synth mode").

EL MUNDO GENUINO-ARDUINO

Sea cual sea el modo de funcionamiento que queramos utilizar (los cuales podemos ir cambiando a lo largo de nuestro sketch sin problemas), primero es necesario descargar e instalar una librería propia (la "GinSing Lib") para poder interactuar convenientemente con el shield GinSing. Está disponible en el apartado de descargas de la página <http://www.ginsingsound.com>.

Centrándonos en el uso de la generación del habla ("voice mode"), podemos consultar la lista de fonemas admitidos dentro de la sección titulada "GSAllophone" del fichero GinSingDefs.h, incluido dentro de la librería anterior. A partir de aquí, podemos estudiar los diferentes códigos de ejemplo incluidos también dentro de la librería anterior, que sirven para mostrar las funcionalidades de cada uno de los modos de funcionamiento del chip. Concretamente, el completo (y largo) código de ejemplo llamado "4_voicemode.ino" nos permite reproducir diferentes frases, incluso cantadas, y aprender el uso del "voice mode" del shield.

Otro shield "parlante" diferente es el llamado "**Voice Shield Slim**" de Spikenzielabs, el cual se basa en el chip ISD4003 de Winbond. En este caso, el sonido no se sintetiza sino que se ha de grabar previamente. Lo interesante de este shield es que permite grabar diferentes unidades de audio identificándolas con un número entero, de forma que podamos combinarlas de la manera que queramos para formar cadenas de unidades más complejas. Es decir, podemos grabar por ejemplo de forma separada la palabra "Yo" (con identificador nº 1), la palabra "Hablo" (con identificador nº 2) y la palabra "Miro" (con identificador nº 3), y entonces hacer que el shield reproduzca la frase "Yo hablo" indicando las unidades 1 y 2 o la frase "Yo miro" indicando las unidades 1 y 3. El sonido emitido, no obstante, tan solo tiene una frecuencia de muestreo de 8KHz y el total de unidades no puede exceder de 4 minutos (240 segundos). Para reproducir el sonido guardado, se puede enchufar un altavoz o amplificador externo a un conector jack de 3,5mm etiquetado como "Audio OUT", pero también se puede utilizar un amplificador interno ya integrado en el shield junto con un altavoz de 15mm a soldar en un espacio reservado.

La manera más sencilla de introducir las unidades de audio en este shield es la siguiente: primero debemos tener guardados en una determinada carpeta de nuestro computador un conjunto de ficheros (en formato WAV o MP3) que representan cada una de las unidades que queremos, y además tener un fichero de texto donde cada una de sus líneas ha de seguir el siguiente formato: *identificador de unidad/tabulador/nombre con extensión del fichero de sonido* (este nombre no ha de tener espacios!). A partir de aquí, deberemos descargarnos e instalar una librería propia llamada "VS Arduino Library", disponible en la página web del producto.

CAPÍTULO 6: ENTRADAS Y SALIDAS

Seguidamente, con el shield acoplado a nuestra placa Arduino, deberemos abrir y grabar en ella un sketch (descargado junto con la librería) llamado "VSLoader.ino". Esto hará que nuestra placa Arduino sea capaz de recibir correctamente la lista de unidades de audio para grabarlas en el chip ISD4003 del shield.

Para realizar esa grabación, nuestra placa Arduino ha de mantenerse conectada vía USB a nuestro computador para controlar el proceso pero además, la entrada de audio del shield (concretamente, un conector jack de 3,5mm etiquetado como "Audio IN") ha de conectarse mediante el cable adecuado a una salida de audio de nuestro computador para recibir los ficheros de sonido propiamente dichos. Una vez realizadas estas conexiones, debemos cerrar el entorno de desarrollo Arduino y ejecutar un programa multiplataforma (descargable de la página web del producto) llamado "VSProgrammer". En él deberemos indicar la carpeta donde se ubican los ficheros de audio que representan las unidades, y el fichero de texto con la lista de todos ellos. Clicando en el botón "Program" se realizará la grabación. Una vez ya grabados, para manejar estos sonidos en nuestros sketches deberemos usar las funciones que proporciona para ello la librería "VS Arduino Library".

DISTRIBUIDORES DE ARDUINO Y MATERIAL ELECTRÓNICO

Existen muchas tiendas online donde podemos adquirir los distintos modelos de placas y shields Arduino. Las variaciones de precio suelen ser de unos pocos euros y el servicio en general es bueno en todas, así que la elección de una u otra se suele basar principalmente en la experiencia personal de cada uno. A continuación, se indican (ni mucho de menos de forma exhaustiva ni completa) alguna de las tiendas más conocidas para adquirir todo lo necesario para nuestros proyectos con Arduino.

El primer sitio donde podemos ir es la tienda oficial de Arduino: <http://store.arduino.cc>. Además de las placas y shields oficiales, desde aquí podemos adquirir otros elementos, como por ejemplo diferentes shields no oficiales, el microcontrolador Atmega328P suelto pero con el bootloader Optiboot ya cargado, el chip controlador de motores L298N, tarjetas microSD, módulos PoE, cables USB de distinto tipo, ristras de pines-macho y pines-hembra de distinta longitud para colocar en shields, etc. También podemos comprar componentes electrónicos básicos para poder montar cualquier circuito, como resistencias, potenciómetros, LDRs, condensadores, LEDs, pulsadores, cables de distinta longitud, pinzas, breadboards de diferente tamaño, etc.

Otro sitio que podemos consultar es <http://arduino.cc/en/Main/Buy>, donde podemos ver un listado exhaustivo de los distintos distribuidores de Arduino oficiales reconocidos, clasificados por países. En todos ellos podremos encontrar las diferentes placas y shields oficiales Arduino, así como el resto de componentes

EL MUNDO GENUINO-ARDUINO

necesarios para realizar cualquier proyecto (desde resistencias, potenciómetros, condensadores, diodos, LEDs, LCDs, transistores, zumbadores, pulsadores, breadboards, cables, multímetros... hasta motores de diversos tipos, material para robótica, módulos XBee, antenas, diferentes tipos de fuentes de alimentación y sus complementos, teclados numéricos, chips de todo tipo, sensores de toda clase, placas y shields no oficiales...), incluyendo también la posibilidad de adquirir kits completos ya preparados con lo esencial. Algunos de los distribuidores que aparecen en la sección "Spain" son:

Electan	(http://www.electan.com)
CanaKit	(http://www.canakit.es)
Bricogeek	(http://www.bricogeek.com/shop)
Cooking-Hacks	(http://www.cooking-hacks.com)
Ardutienda	(http://www.ardumania.es/ardutienda)
Elect. Embaj.	(http://www.electronicaembajadores.com)
Bcncybernetics	(http://www.bcncybernetics.com)
Ro-botica	(http://www.ro-botica.com/arduino.asp), especializados en robótica.

También existen diversos distribuidores para Argentina, Brasil, Chile, Colombia, Costa Rica, Ecuador, México, Panamá y Uruguay.

Aunque si lo que queremos es consultar durante horas y horas el inmensísimo catálogo que presentan los grandes distribuidores de Arduino y electrónica a nivel mundial (y leer la gran cantidad de interesantísimos artículos y tutoriales que muy a menudo ofrecen sobre sus productos, algunos de los cuales están diseñados y fabricados por ellos mismos), a continuación, se muestra una lista (ni mucho menos exhaustiva) de los más importantes. Algunos de ellos han aparecido a menudo a lo largo del libro.

Sparkfun	(http://www.sparkfun.com)
Adafruit	(http://www.adafruit.com)
Makershed	(http://www.makershed.com)

Y también:

DFRobot	(http://www.dfrobot.com)	Freetronics	(http://www.freetronics.com)
Iteadstudio	(http://iteadstudio.com/store)	Seeedstudio	(http://www.seeedstudio.com)
Yourduino	(http://www.yourduino.com)	Fungizmos	(http://store.fungizmos.com)
Modern Device	(http://shop.moderndevice.com)	Cutedigi	(http://www.cutedigi.com)
RSH Electronics	(http://www.rshelectronics.co.uk)	SK Pang	(http://www.skpang.com)
Littlebird Elect.	(http://littlebirdelectronics.com)	Hacktronics	(http://www.hacktronics.com)
Hobbytronics	(http://www.hobbytronics.co.uk)	Mindkits	(http://www.mindkits.com)

APÉNDICE A: DISTRIBUIDORES DE ARDUINO Y MATERIAL ELECTRÓNICO

Cool Components	(http://www.coolcomponents.co.uk)	Oomlout	(http://www.oomlout.co.uk)
Nkcelectronics	(http://www.nkcelectronics.com)	Eio	(http://www.eio.com)
Australian Robotics	(http://australianrobotics.com.au)	Snootlab	(http://www.snootlab.com)
EarthShine Design	(http://www.earthshinedesign.com)	Kineteka	(http://shop.kineteka.com)
EvilMadScience	(http://evilmadscience.com)	Lees	(http://www.leeselectronic.com)
Makerstudio	(http://www.makerstudio.cc)		

Proveedores de Arduino que están más específicamente especializados en componentes para robótica son:

Pololu	(http://www.pololu.com)	Solarbotics	(http://www.solarbotics.com)
RobotShop	(http://www.robotshop.com)	RobotBits	(http://robotbits.co.uk)
TrossenRobotics	(http://www.trossenrobotics.com)	SGBotic	(http://www.sgbotic.com)
ActiveRobots	(http://www.active-robots.com)	RobotGear	(http://www.robotgear.com.au)
Toysdownunder	(http://www.toysdownunder.com)	RobotStore	(http://www.robotstore.com)
RobotElectronics	(http://www.robot-electronics.co.uk)	SuperRobot	(http://www.superrobotica.com)
Juguetronica	(http://www.juguetronica.com)		

Otros proveedores dignos de mención también son: Open Electronics (<http://store.open-electronics.org>), especializado en control remoto y localización, Makerbot (<http://store.makerbot.com>) y RepRap (<http://reprap.org>), especializado en el diseño y construcción de impresoras 3D, o Microcontrollers Shop (<http://microcontrollershop.com>), quien vende todo lo relacionado con microcontroladores Atmel y de otras marcas.

Si estuviéramos buscando un componente electrónico tan extraño que no lo encontráramos en ninguno de los sitios anteriores, después de preguntar en nuestra tienda local más cercana aún podríamos consultar los inabarcables catálogos de los siguientes proveedores más importantes a nivel mundial de componentes eléctricos y electrónicos (la lista no es exhaustiva, ni mucho menos), como por ejemplo:

Mouser	(http://es.mouser.com)	Jameco	(http://www.jameco.com)
Farnell	(http://es.farnell.com)	RS	(http://es.rs-online.com)
Digikey	(http://www.digikey.es)	Newark	(http://www.newark.com)
Conrad	(http://www.conrad.com)	Mpjia	(http://www.mpjia.com)
Maplin	(http://www.maplin.com)	OC	(http://www.onlinecomponents.com)
Verical	(http://www.verical.com)	Vishay	(http://www.vishay.com)
Velleman	(http://www.velleman.eu)	Jaycar	(http://www.jaycar.com)
Satistronics	(http://www.satistronics.com)	Allied El.	(http://www.alliedelec.com)
Future Electronics	(http://www.futureelectronics.com)	Futurlec	(http://www.futurlec.com)
Gateway Catalog	(http://www.gatewaycatalog.com)	BGMicro	(http://www.bgmicro.com)
Sure Electronics	(http://www.sureelectronics.net)	Chip1Stop	(http://www.chip1stop.com)
BPE Solutions	(http://www.bpesolutions.com)	DealExtreme	(http://www.dx.com)
Goldmine	(http://www.goldmine-elec.com)	RapidOnline	(http://www.rapidonline.com)
AllElectronics	(http://www.allelectronics.com)	PartsExpress	(http://www.parts-express.com)

EL MUNDO GENUINO-ARDUINO

Por si fuera poco, también disponemos de dos páginas muy prácticas que nos pueden ayudar a buscar dónde venden un componente determinado y comparar su precio (y stock actual) entre distintos proveedores: <http://www.findchips.com> y <http://www.octopart.com>.

Tambien merece la pena visitar las webs <http://www.electronickits.com> , <http://www.web-tronics.com> ó <http://www.quasarelectronics.com> , donde se puede encontrar prácticamente de todo lo relacionado con robótica, domótica, radio, multimedia, kits educativos, videovigilancia, etc

Kits

De todas formas, lo más cómodo para el principiante de Arduino es adquirir los llamados "kits", formados por una placa Arduino (normalmente, el modelo UNO) más una colección relativamente completa de componentes electrónicos ya preempaquetados. Incluso algunos de ellos vienen con un libro didáctico de proyectos paso a paso. Adquiriendo uno de estos kits, pues, el usuario no se tiene que preocupar buscando individualmente cada elemento que necesita para sus proyectos porque la mayoría de ellos ya los habrá adquirido de golpe con el kit. La mayoría de proyectos que hemos visto en este libro se pueden hacer realidad con cualquiera de estos kits que se listan a continuación:

"Arduino Starter Kit" (de la tienda oficial de Arduino): contiene resistencias de diferentes valores, un termistor, un LDR, un potenciómetro, un diodo, condensadores de diferentes clases, LEDs de diferentes colores, pulsadores, cables, una breadboard, pines para conectar a shields, un transistor bipolar NPN y otro MOS, un zumbador, un chip L293D, un motor DC de 6/9V y un servomotor, un par de sensores de inclinación, un par de optoacopladores y un cable USB A/B para conectar el computador a la placa. Además, incorpora un libro de 170 páginas donde se desarrollan 15 proyectos didácticos, de más sencillos a más complicados; la lista completa se puede leer en <http://arduino.cc/en/Main/ArduinoStarterKit>. Existen dos variantes de este kit: uno que incorpora además de todo lo anterior una placa Arduino UNO y otro en el que no.

"Getting started with Arduino kit" (de Makershed): contiene resistencias de diferentes valores, un par de LDRs, un breadboard, cables, pulsadores, LEDs de diferentes colores, un contenedor para pilas de 9V con conector de 2,1mm, un cable USB y la placa Arduino UNO. Otro kit de Makershed más completo es el **"Ultimate Microcontroller Pack"**, que contiene además de

APÉNDICE A: DISTRIBUIDORES DE ARDUINO Y MATERIAL ELECTRÓNICO

todo lo anterior, dos motores mini-servos, un minimotor DC, un motor de vibración, una pantalla LCD de 16x2, una resistencia sensible a la fuerza (FSR), una caja para guardar los componentes, más modelos de breadboards, condensadores, sensor de inclinación, pulsadores, termistores, interruptores, un diodo, un transistor bipolar NPN, un altavoz, un zumbador y varias ristras de pines.

"Inventor's kit" (de Sparkfun): contiene resistencias de diferentes valores, potenciómetros de varios tipos, un LDR, diodos, LEDs de diferentes colores, pulsadores, un pequeño motor DC, un motor mini-servo, dos transistores bipolares NPN, un relé, un sensor de temperatura, un sensor de flexión, un zumbador, cables, ristra de pines, un breadboard y un soporte para fijarla, un cable USB y la placa Arduino UNO. Contiene además una guía ilustrada de proyectos. Un kit parecido al anterior es el "**Starter kit - Flex**", el cual contiene además un termistor pero no incorpora ni la guía ilustrada, ni los transistores ni los diodos ni los motores ni el relé ni el sensor de temperatura, entre otras diferencias menores. Por otro lado, Sparkfun también distribuye el "**Jumper Wire Kit**", que simplemente es un conjunto extra de cables de diferentes longitudes, ideal para tenerlos siempre a mano por si faltan en nuestros proyectos de prototipado.

"Starter pack" (de Adafruit): contiene resistencias de diferentes valores, potenciómetros, un LDR, LEDs de diferentes colores, pulsadores, cables, un breadboard pequeña, una Protoshield, un adaptador AC/DC a 9V, un contenedor para pilas de 9 V con conector de 2,1mm, un cable USB y la placa Arduino UNO. Un kit parecido al anterior es el "**Budget pack**", pero este no contiene ni el contenedor para pilas ni el adaptador AC/DC ni el Protoshield, entre otras diferencias menores.

"ARDX Experimentation kit" (de Oomlout): contiene resistencias de diferentes valores, potenciómetros, una resistencia sensible a la fuerza (FSR), un LSR, dos diodos, LEDs de diferentes colores, pulsadores, un pequeño motor DC, un motor mini-servo, dos transistores bipolares NPN, un relé, un sensor de temperatura, un zumbador, cables, un breadboard y un soporte para fijarla, un clip para unir una pila de 9V a un jack macho de 2,1mm, un cable USB y la placa Arduino UNO. Contiene además una guía ilustrada de proyectos.

EL MUNDO GENUINO-ARDUINO

Existen muchos otros kits; dignos de mención son: el "**Kit Workshop**" de Ardumania (cuyos componentes más destacados son un sensor de temperatura, un LDR, un motor mini-servo, un zumbador, un transistor MOSFET y un diodo, pero ojo, no incluye ni el cable USB ni la placa); el "**Arduino Starter Kit**", el "**Arduino Sidekick Basic Kit**" y el "**Arduino Lab Kit**" de Cooking-Hacks (a cual más completo, además del "**Components Kit**" que viene sin Arduino); el "**Starter Kit**" de Cutedigi (el cual puede ser adquirido completo o bien por partes llamadas "A", "B", "C", etc.); el "**Arduino Educational Kit**" de Hacktronics; el "**Low-cost Starter Set**" de Yourduino; el "**Starter Kit**" de EarthShine (que contiene un manual de proyectos muy bien escrito, al igual que el "**Arduino Workshop**" de SmileyMicros.com); los "**Kits de iniciación**" de TodoElectronica.com (responsables también de publicaciones periódicas didácticas sobre electrónica muy interesantes), el "**Electronics Tools and Parts Starter Kit**" de CuriousInventor (sin Arduino), los "**Electronics Components Packs 1a**" y "**2a**" de Makershed (sin Arduino), los "**Component Starter Kit**" y "**Resistor Starter Kit**" de Akafugu (sin Arduino), el "**Beginner Parts Kit**" de Sparkfun (sin Arduino), el "**Sidekick Parts Kit v2**" de Seeedstudio (sin Arduino), los "**Arduino Starter Kit**", "**Arduino Advanced Kit**" y "**Arduino DeLuxe Kit**" de Makerstudio (con una placa Arduino clónica no oficial pero más barata), etc.

Por otro lado, en el ámbito general de la electrónica educativa (sin relación con Arduino), son muy interesantes los "**Electronic ProjectLabs**" de <http://www.elenco.com>: kits de componentes electrónicos ya integrados dentro de un panel compacto (fácilmente transportable) que permite el cómodo diseño y la rápida construcción de multitud de circuitos diferentes (muchos de los cuales vienen explicados y desarrollados en sendas guías impresas, incluidas en el producto).

B

CÓDIGOS IMPRIMIBLES DE LA TABLA ASCII

Código numérico	Carácter	Código numérico	Carácter	Código numérico	Carácter
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	\$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(72	H	104	h
41)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m

EL MUNDO GENUINO-ARDUINO

46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[123	{
60	<	92	\	124	
61	=	93]	125	}
62	>	94	^	126	~
63	?	95	_	127	DEL
64	@	96	`		

RECURSOS PARA SEGUIR APRENDIENDO

Una vez finalizada la lectura de este libro, iesto no ha hecho más que empezar! Las posibilidades que se abren a partir de ahora son realmente casi infinitas. No obstante, el lector necesitará otros recursos de información para progresar. Así pues, para evitar el estancamiento, a continuación, ofrecemos una lista de recursos recomendados (tanto en papel como en formato web) para profundizar en el conocimiento de las diversas áreas tratadas en el libro.

Plataforma Arduino

El mejor sitio para continuar aprendiendo todo sobre Arduino es, lógicamente su página web oficial. Lo más interesante es tal vez su "Playground" (<http://www.arduino.cc/playground>), wiki editable por la comunidad donde se muestran multitud de trucos y tutoriales prácticos de la más diversa índole. Otra fuente de información (aunque mucha de ella ya ha sido expuesta en este libro) es el apartado de "Hacking" de la misma web oficial (<http://www.arduino.cc/en/Hacking>). También existe, dentro de la web oficial, un apartado a enlaces externos con multitud de información sobre Arduino (<http://www.arduino.cc/en/Tutorial/Links>).

La ayuda que podemos recibir no se acaba aquí, ni mucho menos: también podemos formar parte de la amplia comunidad Arduino existente en el mundo a través del foro oficial (<http://forum.arduino.cc>). Desde allí podremos ayudar y ser ayudados en cualquier tema relacionado con Arduino: existen muchas categorías, que van desde consultas sobre electrónica básica, programación básica, microcontroladores, problemas en la instalación y uso del IDE, tipos de sensores,

EL MUNDO GENUINO-ARDUINO

tipos de actuadores, maneras de comunicar la placa –red, serie, etc.–... hasta consultas relacionadas con ámbitos de uso, como la robótica, la domótica, el diseño interactivo, educación, ciencia, uso de textiles... pasando por consultas sobre posibles mejoras al hardware y al software Arduino, noticias sobre eventos y talleres, etc. Incluso existe un apartado específico para la comunidad hispanohablante. Es decir, resumiendo: cualquier persona interesada en Arduino debe consultar el foro oficial si quiere acceder a una ingente cantidad de información de valor incalculable, aportada gracias a la cooperación desinteresada de toda la comunidad, que hace que Arduino sea un proyecto cada vez más y más grande. ¡No olvides colaborar y aportar tu propio granito de arena!

No está de más comentar también la existencia tanto de la cuenta de Twitter oficial de Arduino (@arduino) como del blog oficial de Arduino (<http://blog.arduino.cc>), que nos permitirán estar al día de los proyectos más interesantes que surjan dentro de la comunidad, así como de las novedades oficiales que vayan apareciendo.

Finalmente, en la dirección <http://www.arduino.cc/en/Main/ContactUs> se listan las direcciones de correo electrónico utilizadas por el Arduino Team para recibir sugerencias o consultas sobre algún aspecto de Arduino que no haya quedado suficientemente aclarado en las páginas anteriores.

Respecto a la información bibliográfica existente sobre Arduino, es un hecho que cada vez es más extensa y variada. Un rápido vistazo al catálogo de la librería online Amazon, por ejemplo nos revelará multitud de títulos: algunos más enfocados a la vertiente hardware de Arduino y otros más a la vertiente de programación, otros centrados en ser un compendio de proyectos y otros en ser meras introducciones, otros más especializados en el estudio de la comunicación de sensores en redes inalámbricas y otros en la comunicación con sistemas Android, otros relacionados con la robótica y otros más en la domótica, o en el textil electrónico, etc. En este sentido, para poder elegir el libro que realmente necesitemos, recomiendo consultar su índice y así confirmar si nos ofrece lo que buscamos. También es importante fijarse en la fecha de edición, ya que fechas demasiado antiguas incorporarán códigos obsoletos.

Electrónica general

Para aprender electrónica, la mejor opción es adquirir un libro especializado en la materia. Para iniciarse recomiendo los siguientes:

APÉNDICE C: RECURSOS PARA SEGUIR APRENDIENDO

- | | |
|-----------------------------------|--|
| *"Make: Electronics" | Charles Platt. <i>Make; 1ed (2009)</i> |
| *"Getting Started In Electronics" | Forrest M. Mims III. <i>Master Publishing, Inc. (2003)</i> |
| *"Electronics for Dummies" | Cathleen Shamieh. <i>For Dummies;2ed (2009)</i> |

De un nivel intermedio podemos destacar:

- | | |
|---|--|
| *"Complete Electronics Self-Teaching Guide" | Earl Boysen. <i>Wiley;3ed (2012)</i> |
| *"Practical electronics for inventors" | Paul Scherz. <i>McGraw-Hill; 2ed (2006)</i> |
| *"Circuitbuilding for dummies" | H.Ward Silver. <i>For Dummies; 1ed (2008)</i> |
| *"Electronic projects for dummies" | Earl Boysen. <i>For Dummies; 1ed (2006)</i> |
| *"Teach Yourself Electricity and Electronics" | Stan Gibilisco. <i>McGraw-Hill; 5ed (2011)</i> |
| *"Beginner's Guide to Reading Schematics" | Robert J. Traister. <i>TAB Books; 2ed (1991)</i> |
| *"Lessons in electric circuits" | Tony R. Kuphaldt (2006-2010) |

El último libro mencionado en realidad son varios volúmenes descargables gratuitamente de <http://openbookproject.net/electricCircuits> Viene acompañado de un conjunto de preguntas, disponibles en <http://www.ibiblio.org/kuphaldt/socratic>.

Y de un nivel bastante más avanzado:

- | | |
|--|--|
| *"The Art of Electronics" | P. Horowitz. <i>Cambridge U.P;2ed (1989)</i> |
| *"The Circuit Designer's Companion" | Tim Williams. <i>Newnes;2ed (2005)</i> |
| *"Practical Introduction to Electronic Circuits" | M.H.Jones. <i>Cambridge U.P;3ed (1996)</i> |

Si, no obstante, se desea utilizar recursos online, una página web que contiene tutoriales de electrónica y circuitos de ejemplo interesantes es <http://www.kpsec.freeuk.com>, y también <http://www.radio-electronics.com>. Una web que contiene muchos ejemplos de circuitos es <http://www.extremecircuits.net>. Si se quieren conocer más recursos online, una web con muchos enlaces es <http://www.dapj.net/hobby/educational>.

Proyectos

Existen varios portales web que diariamente recopilan proyectos muy interesantes publicados en blogs personales. En realidad, muchos proyectos no están relacionados exclusivamente con Arduino, sino que muestran proyectos autónomos de robótica, domótica, control remoto, multimedia, impresión 3D..., pero consultar estos portales es una buena manera para estar al día de las ideas que van surgiendo en la comunidad y para tomar buena nota de los trucos y conocimientos empleados por mucha gente. Está claro que para aprender procedimientos frescos y adquirir

EL MUNDO GENUINO-ARDUINO

ideas nuevas, lo mejor es estudiar proyectos de otras personas, y estos portales nos lo permiten. Algunos de ellos son:

Instructables (<http://www.instructables.com>)

Makezine (<http://blog.makezine.com/arduino>)

MakeProjects (<http://makeprojects.com/Topic/Arduino>)

HackADay (<http://hackaday.com/category/arduino-hacks>)

HackNMod (<http://hacknmod.com>)

Dangerous Prototypes (<http://dangerousprototypes.com>)

Electronics Lab (<http://www.electronics-lab.com/blog>)

BricoGeek (<http://www.bricogeek.com>)

Embedds (<http://www.embedds.com>)

Sección de proyectos Jameco (<http://www.jameco.com/Jameco/PressRoom/diy.html>)

Sección de proyectos Arduino (<http://arduino.cc/playground/Projects/ArduinoUsers>)

HLT (Proyectos: <http://highlowtech.org/?cat=5>

Materiales:<http://highlowtech.org/?cat=21>

Tutoriales: <http://highlowtech.org/?cat=20>)

Algunos distribuidores (como Sparkfun, Adafruit, Makershed, Bricogeeek, Cooking-Hacks, y más) incorporan dentro de sus páginas oficiales secciones donde informan puntualmente de las novedades y los eventos más destacables dentro del mundo de la computación física, siendo pues su consulta muy recomendable para estar al día en este ámbito.

ÍNDICE ANALÍTICO

#

#define, 174
#include, 219

A

abrir el circuito, 13
ACM, 136
actuador, 61
Adafruit GFX, 237, 244, 247, 248, 249, 256, 257
Adafruit RGB, 237
Adafruit Touch Screen, 249
adaptador AC/DC, 33
adaptador USB-Serie, 105
adaptadores no regulados, 34
adaptadores regulados, 33
aislante, 2, 5
alimentación de desvío, 49
amperio, 3
amperios-hora, 24
amplificador operacional, 245
Android, 103
ánodo, 41
ánodo común, 253
Arduino, 63, 69
Arduino IR, 429
Arduino UNO, 71

ARM, 114
array, 168
ASCII, 179
ATmega16U2, 89
ATmega2560, 102
ATmega328P, 74
ATmega32U4, 113
Atmel, 64
ATtiny. Véase tinyAVR
ATxmega. Véase XMEGA
AVR, 64
AVR109, 84
avrdude, 134, 141
avr-libc, 134

B

backpack, 232
batería, 21
baudio, 147
bias, 486
bit, 76
BJT, 50
boolean, 162
bootloader, 76, 83
botón de reinicio, 100
bounce, 316
breadboard, 61
BSSID, 553

EL MUNDO GENUINO-ARDUINO

buffer, 181, 182, 274
by-pass. Véase alimentación de desvío
byte, 76, 164

C

C, 134, 151
C++, 134, 151
caída de potencial, 2
calibración, 408
capacidad, 25
capacidad, 47
casting, 171
Caterina, 84
cátodo, 41
cátodo común, 253
CDC, 136
CdS. Véase fotoresistores
cerrar un circuito, 12

Ch

char, 162
char*, 170
chip, 63, 62
chipKIT, 152

C

ciclo de trabajo, 375
colores de una resistencia, 36
ColorLCDShield, 238
compilación, 149
comunicación "paralela", 79
comunicación "serie", 79
concatenar, 193
condensador, 47
condensadores de filtro, 49
condensadores polarizados, 48
condensadores unipolares, 48
conductor, 1, 5
conectores USB, 32
conexión en paralelo, 13
conexión en serie, 13
constante de tiempo, 434, 487
conversor analógico/digital, 90

corriente alterna, 3
corriente continua, 3
corriente de paro, 277
CPU, 62
Creative Commons, 68

D

Darlington, 286
demodulador, 420
detector de envolvente, 487
DeuLigne, 237
DFU, 121
DHCP, 494
diferencia de potencial, 2
diodo, 41
DIP, 72
diseño de referencia, 101
divisor de tensión, 16
double, 168

E

E/S, 63
electricidad, 1
electrón, 1
ENC28J60, 549
encapsulado, 72
entorno de desarrollo, 65
EthernetUDP, 502
exFAT, 261

F

factor de reducción, 287
faradios, 47
FAT32, 261
FET, 50
fijador de nivel, 488
FILE_READ, 265
FILE_WRITE, 265
filtro "pasa-altos", 380
filtro pasa-altos, 489
Firmata, 221
float, 167
fly-back, 278, 358

ÍNDICE ANALÍTICO

fotointerruptor, 418

fotorresistores, 41

Frecuencia, 10

frecuencia de corte, 380

FSR, 41

FT232RL, 89, 106

FTDI, 89

fuente de alimentación, 6, 21, 61

full duplex, 81

función, 199

G

gcc, 133

gearhead, 279

gestor de arranque. Véase bootloader

GLCD, 224

global, 161

GPIO, 90

GPL, 67

ground. Véase tierra

H

half duplex, 80

Hardvard, 77

hardware libre, 68

HD4478, 232

HD44780, 219

hercio, 10

hexadecimal, 166

host USB, 102

I

I²C, 80

IBSS, 553

IC. Véase chip

ICSP, 96

IEC, 36

IEEE802.3, 104

impedancia, 381

importar librerías, 144

int, 165

Intel Hex Format, 84

intensidad, 3

interrupciones, 94

IOREF, 95

ISM, 564

J

Java, 133

L

L293, 280

L298, 280

LAN, 104

LCD Assistant, 244

LDO. Véase regulador de voltaje

LDR. Véase fotoresistores

LED, 44

LED RGB, 332

lenguaje Arduino, 65

lenguaje de programación, 65

Ley de Ohm, 5

LGPL, 67

librerías, 70

LiPo, 22, 28

LiveGraph, 450

LM386, 491

local, 161

long, 165

low-active, 420

M

mapear, 189

Maple, 152

masa. Véase tierra

media, 439

megaAVR, 74

Memoria EEPROM, 77

Memoria Flash, 76

Memoria SRAM, 77

memorias SD, 77

Menwiz, 238

microcontrolador, 62

midspan, 107

milicandelas, 45

MISO, 81

EL MUNDO GENUINO-ARDUINO

módulo, 316
MOSI, 81
motor DC, 123
motor paso a paso., 122
MSGEQ7, 483
MSP430, 152
MultiMediaCard, 262
multímetro digital, 74

N

nivel de línea, 484
NPN, 51
NTC, 434

O

objeto, 176
octava, 376
ohmio, 5
operador de asignación, 206
operadores aritméticos, 193
operadores booleanos, 206
operadores compuestos, 215
operadores de comparación, 203
Optiboot, 83
optoacoplador, 419
osciladores de cristal, 99
OSHD, 69
overflow, 167

P

par, 278
PBx, 75
PCB, 64
PCx, 75
PDx, 75
perfboard, 61, 64
perfiles de dispositivo, 564
Periodo, 10
Phi_big_font, 232
Phi_prompt, 237
PIC, 74
PID, 137
pila, 21

pila "TCP/IP", 104
piroelectricidad, 467
placa "breakout", 73
PNP, 51
polo, 1
potencia, 6
potenciómetro, 38
pre-amplificador, 484
Processing, 65
programa, 129
programador ISP, 84, 96
proto shields, 126
protoboard. Véase breadboard
proxy serie-red, 550
puente H, 280, 286, 361
pull-up, 297
pulsador, 54
puntero, 170
punto-Q, 490
PWM, 92

R

R-2R, 346
rectificador, 33, 42
reductor, 279
reed switch, 399
registros, 78
regulador de tensión. Véase regulador de voltaje
regulador de voltaje, 33
RESET, 95
resistencia, 4
resistencia "pull-down", 18
resistor, 35
resonadores cerámicos, 99
rpm, 278
RS-232, 88
RTC, 273
ruido, 9, 49, 337
RX, 94, 176
rxtx, 133

ÍNDICE ANALÍTICO

S

SAM3X8E, 114
SCK, 81
SCL, 79
screw shields, 127
SDA, 79
SdFat, 272
SdfatLib, 220
SDHC, 261
SDIO, 261
SDSC, 261
SDXC, 261
sensor, 61
señal analógica, 8
señal aperiódica, 10
señal binaria, 7
señal digital, 7
señal periódica, 10
SerialUSB, 187
SerLCD, 235
shield, 116
sistema electrónico, 61
sketch, 129
SMD, 72
SMT, 73, 125
software libre, 67
SPI, 80
splitter PoE, 109
SS, 81
Steinhart-Hart, 436
STK500, 83
stripboard, 61, 64

THT, 125
tierra, 12
Timer1, 93
Tinkerkit, 281
tinyAVR, 74
torque. Véase par
transformador, 33
transistor, 50
TTL-UART, 88, 94, 176, 185
TWI, 79
TX, 94, 176

U

ULN2003, 286
ULN2803, 291
unidad C, 25
unsigned long, 165
TFT, 252

V

Valor instantáneo, 11
Valor medio, 11
vatio, 6
VCC, 75
VID, 137
Vin, 87
voltaje, 2
voltaje de referencia, 91
voltio, 3
Von Neumann, 78

W

W5100, 104, 117
Wiring, 64, 152
word, 165

X

XBee, 110, 119
XMEGA, 74