

Unidad “Control de periféricos”
Comunicación serie entre PC y placa Arduino

Gonzalo F. Perez Paina

19 de junio de 2021

Índice general

1. Comunicación serie y estándar RS-232	1
1.1. Introducción	1
1.2. Comunicación paralela y serie	1
1.2.1. Transmisión síncrona vs. asíncrona	3
1.2.2. Comunicación Half-Duplex y Full-Duplex	4
1.2.3. Ejemplos de comunicación paralela y serie	4
1.3. El estándar RS-232	5
1.3.1. Características eléctricas	7
1.3.2. Características funcionales	8
1.3.3. Características mecánicas	8
1.3.4. Receptor/transmisor asíncrono universal, UART	9
1.4. Puerto serie en una PC	10
1.4.1. Acceso a dispositivos	11
1.4.2. Dispositivos TTY	12
1.4.3. Comunicación entre dispositivos TTY	12
1.5. Programación del puerto serie de la PC	14
1.5.1. Programa ejemplo	14
1.5.2. Configuración de la terminal – estructura termios	15
1.5.3. Módulo para configuración del puerto serie	15
2. Programación del microcontrolador ATmega328	19
2.1. Introducción	19
2.2. La placa Arduino UNO	19
2.3. El microcontrolador ATmega328	20
2.3.1. Memoria	21
2.3.2. Puerto digital de entrada/salida	22
2.3.3. Puerto serie USART	25
2.4. Programación del ATmega328	30
2.4.1. Programación con el IDE Arduino	31
2.4.2. Programación sin el IDE Arduino	35
2.5. Programación de los puertos digitales	37
2.5.1. Programación de salidas digitales	37
2.5.2. Programación de entradas digitales	38
2.5.3. Constantes simbólicas y macros	40
2.6. Programación de la USART	42

Índice general

2.6.1. Configuración de la USART	42
2.6.2. Transmisión de datos	43
2.6.3. Recepción de datos	44
3. Comunicación entre PC y placa Arduino	49
3.1. Conexión entre la placa Arduino UNO y la PC	49
3.2. Ejemplo de comunicación serie	50
3.2.1. Transmisión	50
3.2.2. Recepción	50
3.3. Actividad práctica	54

Capítulo 1

Comunicación serie y estándar RS-232

1.1. Introducción

En informática, la comunicación o transmisión de datos entre dispositivos se puede realizar de forma paralela o serie. La comunicación paralela o en paralelo permite transmitir varios bits (del inglés *binary digit* o dígito binario) de forma simultánea. En contrapartida, la comunicación serie o serial transmite un único bit a la vez de forma secuencial. La principal diferencia entre un canal de comunicación paralelo y uno serie es la cantidad de conductores eléctricos (cables) en la capa física para la transmisión de los bits; por ejemplo, en la comunicación en paralelo se necesitan tantos conductores como bits se desea transmitir.

1.2. Comunicación paralela y serie

La transmisión de datos permite comunicar equipos electrónicos tales como PC, PLC (Controladores Lógicos Programables), instrumentos de laboratorios (multímetros, osciloscopios, etc.), placas de desarrollo de sistemas embebidos (Arduino, Raspberry Pi, etc.); como así también entre componentes internos de una PC.

Algunos ejemplos de sistemas de comunicación paralela son las interfaces (buses) que permiten interconectar componentes internos de una PC tales como:

- ISA: Industry Standard Architecture.
- ATA/IDE: Advanced Technology Attachment/Integrated Drive Electronics.
- SCSI: Small Computer System Interface.
- PCI: Peripheral Component Interconnect.

En cuanto la comunicación de la PC con dispositivos externos se pueden mencionar:

- IEEE-1284 (Standard Signaling Method for a Bi-directional Parallel Peripheral Interface for Personal Computers): incluido en la primeras PC de IBM y luego estandarizado por el Institute of Electrical and Electronics Engineers (IEEE). Brinda

1. Comunicación serie y estándar RS-232

una comunicación de “alta velocidad” y bidireccional entre una PC y un dispositivo externo.

- IEEE-488: originalmente desarrollados por Hewlett-Packard (HP-IB: Hewlett-Packard Instrument Bus o GPIB: General-Purpose Instrumentation Bus) para conectar dispositivos de testeo y medición (por ejemplo multímetros, osciloscopios, etc.) con una PC. Luego estandarizado por el IEEE.

Por el lado de la comunicación serie se pueden mencionar los estándares RS-232, RS-422, RS-485 y protocolos como el SPI, I²C, 1-Wire, entre otros.

- RS-232/EIA-232 (Recommended Standard/Electronic Industries Alliance): define las señales que se conectan entre un DTE (Data Terminal Equipment o Equipo Terminal de Datos) como un terminal de computadora, y un DCE (Data Communication Equipment o Equipo de Comunicación de Datos) como un módem.
- RS-485/EIA-485: estándar de comunicaciones en bus diferencial de la capa física del Modelo OSI (Open System Interconnection).

Otras interfaces de comunicación serial son:

- SPI (Serial Peripheral Interface): especificación de comunicación serie usado principalmente para la transferencia de información entre circuitos integrados en equipos electrónicos.
- I²C (Inter-Integrated Circuit): bus serie de datos desarrollado por Philips Semiconductors (hoy NXP Semiconductors, parte de Qualcomm), utilizado principalmente para la comunicación entre diferentes partes de un circuito, por ejemplo, entre un controlador y circuitos periféricos integrados.
- 1-Wire: protocolo de comunicaciones diseñado por Dallas Semiconductor, basado en un bus, un maestro y varios esclavos de una sola línea de datos.

La figura 1.1 muestra una representación esquemática de dispositivos conectados en paralelo (figura 1.1a) y en serie (figura 1.1b). Como se observa, en la comunicación paralela se necesitan tantas conexiones como bits se quieran transmitir, mientras que para una comunicación serie en un solo sentido se necesitan solo una. Cabe aclarar que esta es una representación esquemática y no un diagrama eléctrico en cuyo caso sería necesario agregar también la señal de referencia (GND) entre ambos dispositivos.

En cuanto a los sistemas embebidos, la conexión más comúnmente utilizada de dos dispositivos en paralelo es entre una placa de desarrollo como Arduino y una pantalla LCD (Liquid-Crystal Display) ya sea esta de caracteres o gráfica. Mientras que la transmisión de datos en serie se utiliza en general en módulos de comunicación inalámbrica Bluetooth, módulos GPS (Global Positioning System), GSM (Global System for Mobile communications), etc.

La tabla 1.1 muestra la comparación entre algunas de las características de la comunicación serie y paralela.

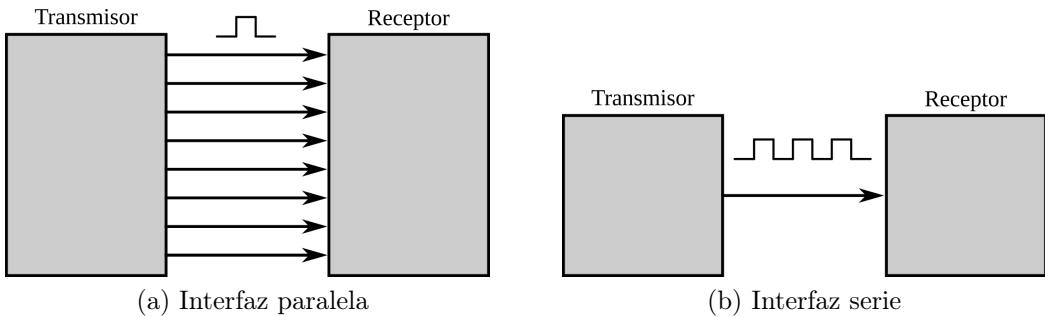


Figura 1.1: Comunicación paralela vs. serie.

Característica de transmisión	Serie	Paralela
Descripción	Se envía un bit tras de otro, uno a la vez	Todos los bits de un solo carácter se transmiten simultáneamente
Velocidad comparativa	Más lenta	Más rápida
Limitación de distancia	Más lejos	Más cerca
Aplicaciones	Entre dos computadoras, desde computadoras hasta dispositivos externos, redes de área local (LAN) y amplia (WAN)	Dentro de una computadora a lo largo de los buses de la misma, entre un controlador de unidad y un disco duro
Cables	Todos los bits viajan por un solo cable, un bit a la vez	Cada bit viaja por su propio cable simultáneamente con otros bits

Tabla 1.1: Comparativa entre transmisión paralela vs. serie.

1.2.1. Transmisión síncrona vs. asíncrona

Cuando dos dispositivos se comunican, intercambian algún tipo de señal detectable que representa los datos y a menos que los dispositivos estén tan ocupados que nunca dejen de hablar, habrá un momento en que no se enviará ninguna señal a través del canal de comunicación. Durante estos tiempos “muertos” no se envían datos a través del canal. Luego, cuando haya datos para enviar, el dispositivo de envío comenzará a enviar señales, por lo que debe haber una forma para que el dispositivo de destino sepa cuándo comenzar a buscar/leer datos: los dos dispositivos deben establecer y mantener algún tipo de sincronización entre ellos para que las señales se produzcan, transmitan y detecten con precisión. Hay dos alternativas principales para establecer y mantener la sincronización para el muestreo de las señales y se conocen como transmisión asíncrona y síncrona.

En la comunicación asíncrona, la sincronización se restablece con la transmisión de cada carácter mediante el uso de bits de inicio y parada que, dependiendo de la tecnología utilizada, puede haber 1, 1,5 o 2 bits de parada. En la comunicación síncrona, la sincronización se mantiene mediante un bit especial en cada bloque de datos proporcionado por una señal adicional de reloj suministrada por los dispositivos de comunicación. Debido a

1. Comunicación serie y estándar RS-232

la menor cantidad de bits utilizados para mantener la sincronización en la comunicación síncrona, esta resulta más eficiente que las comunicaciones asíncronas.

La figura 1.2 muestra de forma esquemática la comunicación serie síncrona y asíncrona entre dos dispositivos. En ambos casos, los dos dispositivos pueden transmitir y recibir información. En la figura 1.2a la comunicación es síncrona, por lo que además de las señales de transmisión y recepción de datos, se incluye también una señal de reloj o clock; mientras que en la figura 1.2b al ser una comunicación asíncrona, esta señal de reloj no es necesaria.

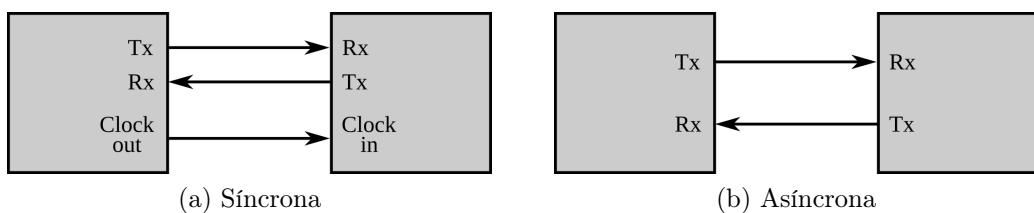


Figura 1.2: Comunicación síncrona vs. asíncrona

1.2.2. Comunicación Half-Duplex y Full-Duplex

Las sesiones de comunicación de datos son en general de naturaleza bidireccional, incluso si el objetivo de la comunicación es enviar un archivo desde el remitente al destino, alguna comunicación debe ir desde el destino de regreso al remitente. Estos datos en dirección inversa incluyen por ejemplo información de corrección y detección de errores. Existen dos maneras de manejar este tipo de tráfico bidireccional: Full-Duplex y Half-Duplex.

En una comunicaciones Full-Duplex, ambos dispositivos pueden transmitir al mismo tiempo, esto es análogo a una conversación en persona. En una Half-Duplex, uno de los dispositivos puede oír o hablar en un momento determinado, lo cual es similar a una conversación que tiene lugar a través de un walkie-talkie (solo puede hablar aquel que presione el botón). De manera similar, cuando dos dispositivos de comunicación de datos se comunican en un entorno Half-Duplex, un dispositivo debe ser el transmisor y el otro el receptor. Para permitir la comunicación bidireccional, periódicamente tienen que invertir roles.

Cuando la comunicación es unidireccional, se conoce con el nombre de Simplex, como es el caso del esquema de la figura 1.1.

1.2.3. Ejemplos de comunicación paralela y serie

La figura 1.3 muestra la conexión entre una placa Arduino UNO y una pantalla (display) o LCD (Liquid Crystal Display) de caracteres de 16×2 , utilizando 8 bits de datos en paralelo. Estas pantallas se conectan en paralelo utilizando 8 o 4 bits de datos donde, para el último caso, se debe dividir el byte a escribir en 2 nibbles y realizar 2 operaciones de escritura.

Observando de cerca estas pantallas se pueden ver los pequeños rectángulos (píxeles) que forma cada carácter. Cada uno de estos rectángulos es una cuadrícula de 5×8 píxeles. Las pantallas LCD de caracteres vienen de diferentes tamaños y colores: por ejemplo, 16×1 , 16×2 , 20×4 , con texto blanco sobre fondo azul, con texto negro sobre verde entre otras variantes.

Como se observa en la figura, además de los bits de datos se necesitan de otras señales o bits de control, los cuales son: RS (Register Select), E (Enable) y R/W (Read/Write).

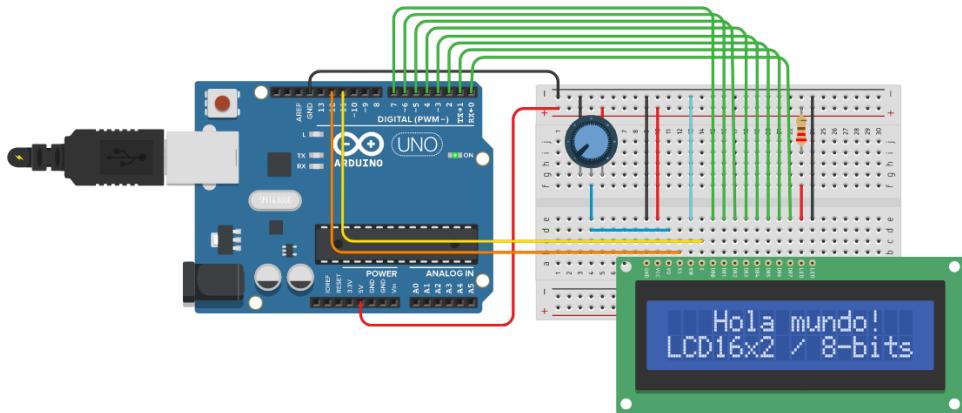


Figura 1.3: Conexión en paralelo de 8-bits entre placa Arduino UNO y LCD de caracteres.

La figura 1.4 muestra la conexión entre una placa Arduino UNO con diferentes módulos mediante comunicación serie (interfaz UART-TTL). En la figura 1.7a se conecta un módulo de comunicación inalámbrica Bluetooth modelo HC-05, y en la figura 1.7b un módulo GPS (Global Positioning System) modelo NEO-6MV2. Como se verá más adelante, para este tipo de comunicación serie asíncrona se necesitan conectar dos bits de datos (Tx y Rx) además de la referencia común de tensión (GND).

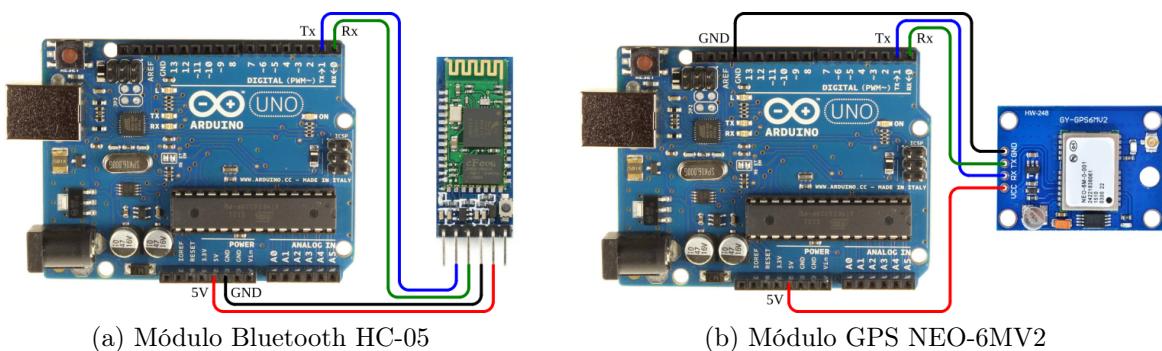


Figura 1.4: Conexión en serie de placa Arduino UNO con módulos Bluetooth y GPS.

1.3. El estándar RS-232

Debido a su relativa simplicidad en el hardware, en comparación con la interfaz en paralelo, las comunicaciones en serie se utilizan ampliamente en la industria electrónica.

1. Comunicación serie y estándar RS-232

Uno de los estándares de comunicación en serie más utilizado es sin duda la especificación EIA/TIA-232-E. Este estándar, que ha sido desarrollado por la *Electronic Industry Association* (Asociación de la Industria Electrónica) y la *Telecommunications Industry Association* (Asociación de la Industria de las Telecomunicaciones), se conoce más popularmente simplemente como *RS-232*, donde “RS” significa *Recommended Standard* (Estándar Recomendado).

El nombre oficial del estándar EIA/TIA-232-E es *Interface Between Data Terminal Equipment and Data Circuit-Termination Equipment Employing Serial Binary Data Interchange* (Interfaz entre equipos terminales de datos y equipos de terminación de circuitos de datos que emplean intercambio de datos binarios en serie). Aunque pueda parecer intimidante, el estándar se refiere simplemente a la comunicación de datos en serie entre un sistema host (DTE, Data Terminal Equipment) y un sistema periférico (DCE, Data Circuit-Terminating Equipment o Data Communication Equipment). Gran parte de la terminología RS-232 refleja su origen como estándar para las comunicaciones entre un terminal de computadora (PC) y un módem externo.

Para comprender el uso de la palabra *terminal*, es necesario remontarse al tiempo cuando las computadoras eran del tamaño de habitaciones enteras. Dichas computadoras tenían pocas formas de interactuar con el exterior, las tarjetas perforadas y los rollos de cinta de papel eran una de esas interfaces, pero también existía lo que se conocía como una terminal que se usaba para ingresar y recuperar datos. Estas terminales llegaron en muchos factores de forma, pero pronto comenzaron a parecerse a lo que se convertiría en sus descendientes de computadoras personales.

En este sentido, una terminal contiene un teclado, una pantalla, un puerto de comunicaciones para acceder a una computadora remota y poco más. Un enlace RS-232 conecta el terminal a un módem, que a su vez accede a las líneas telefónicas que se conectan a la computadora remota. Las PC con módems e interfaces de red han hecho que este tipo de conexión de terminal sea casi obsoleto. Además, en la actualidad resulta más frecuente el uso del puerto RS-232 para conectar una PC a un sistema embebido o bien dos sistemas embebidos entre sí.

En el estándar RS-232 se hace referencia al DTE o equipo terminal de datos y DCE o equipo de comunicación de datos, en el cual las señales y sus funciones reciben su nombre desde la perspectiva del DTE. Por ejemplo, Tx (transmisión de datos) es una salida en el DTE y una entrada en el DCE, mientras que Rx (recepción de datos) es una entrada en el DTE y una salida en el DCE. La figura 1.5 muestra la aplicación típica de la interfaz RS-232 en una comunicación utilizando módems.

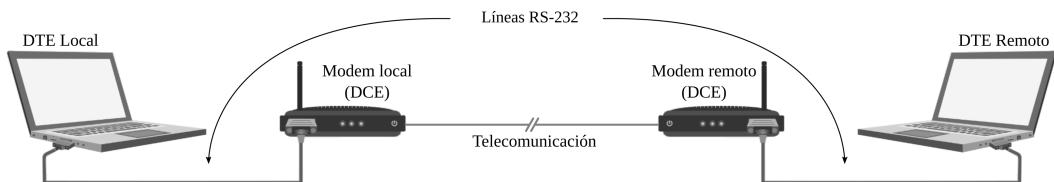


Figura 1.5: Interfaz entre DTE y DCE.

El RS-232 es un estándar “completo”, lo que significa que el estándar se propone garantizar la compatibilidad entre el host y los sistemas periféricos especificando: 1) niveles

comunes de voltaje y señal, 2) configuraciones de pines y cableado y 3) una cantidad mínima de información de control entre el host y los sistemas periféricos.

1.3.1. Características eléctricas

La sección de características eléctricas del estándar RS-232 incluye especificaciones sobre los niveles de tensión, tasa de cambio de niveles de las señal e impedancia de la línea. El estándar RS-232 original se definió en 1962, previo a los días de la lógica TTL (Transistor-transistor Logic), por lo que no es de extrañar que no utilice niveles lógicos de 5 voltios. En cambio, se define un nivel alto de salida con tensiones entre +5 y +15 voltios y un nivel bajo entre -5 y -15 voltios. Los niveles lógicos de entrada se definieron para proporcionar un margen de ruido de 2 voltios, por lo tanto un nivel alto para la entrada debe estar comprendido entre +3 a +15 voltios y un nivel bajo entre -3 a -15 voltios. Es necesario tener en cuenta que para la comunicación RS-232 un nivel bajo (-3 a -15 voltios) se define como un 1 lógico y se conoce históricamente como *mark* (marca), mientras que un nivel alto (+3 a +15 voltios) se define como un 0 lógico y se denomina *space* (espacio). La figura 1.6 muestra los niveles lógicos definidos por el estándar RS-232.

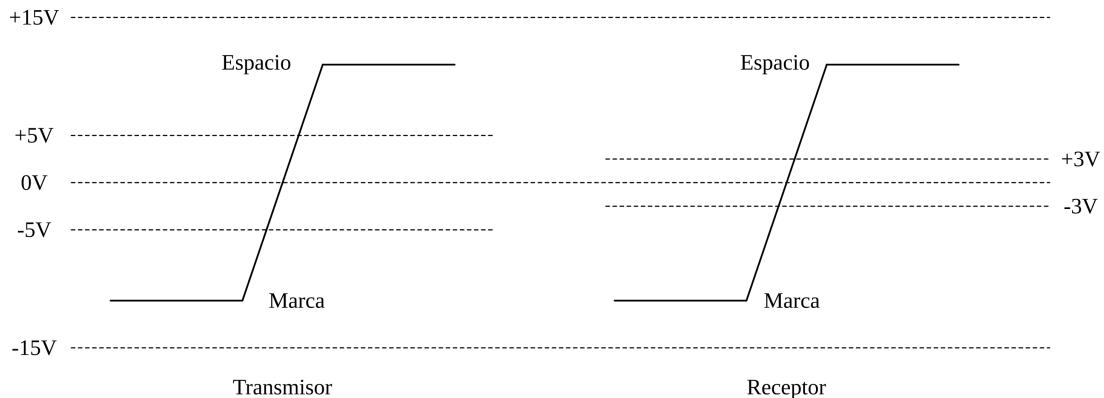


Figura 1.6: Especificaciones de niveles lógicos del RS-232.

El estándar RS-232 también limita la velocidad máxima de respuesta (slew-rate) en la salida de los circuitos para ayudar a reducir la probabilidad de interferencias entre señales adyacentes (cross-talk). Cuanto más lento sea el tiempo de subida y bajada, menor será la posibilidad de interferencia cruzada. Teniendo esto en cuenta, la velocidad de respuesta máxima permitida es de $30V/\mu s$. Además, el estándar define una velocidad máxima de datos de 20Kbits/seg., con el propósito también de reducir la posibilidad de interferencias cruzadas.

Para el estándar RS-232 original se especificó una longitud de cable máxima de 15 metros, pero esta parte de la norma se modificó en la revisión "D" (EIA/TIA-232-D), en la cual en lugar de la longitud máxima del cable se especificó una carga capacitiva máxima de 2500pF, que resulta en una especificación más adecuada. La longitud máxima del cable está determinada por la capacitancia por unidad de longitud del cable que se proporciona en las especificaciones propias del cable.

La mayoría de los sistemas actuales no funcionan con niveles de voltaje del estándar RS-232, por lo que se necesita de una conversión de nivel para implementar la comunica-

1. Comunicación serie y estándar RS-232

ción serie, la cual se lleva a cabo mediante circuitos integrados especiales. Estos circuitos integrados (CI) suelen tener drivers de salida que generan los niveles de voltaje requeridos por estándar RS-232 y receptores que pueden recibir niveles de voltaje RS-232 sin sufrir daños.

1.3.2. Características funcionales

El segundo aspecto que está cubierto por el estándar hace referencia a las características funcionales de la interfaz. Esto significa que el RS-232 define la función de las diferentes señales que se utilizan en la interfaz. Estas señales se dividen en cuatro categorías diferentes: referencia, datos, control y sincronización. El estándar proporciona una gran cantidad de señales de control y admite un canal de comunicaciones primario y secundario. Afortunadamente, pocas aplicaciones, si las hay, requieren todas estas señales definidas en el estándar. Por ejemplo, se necesitan de solo ocho señales para un módem típico y algunas aplicaciones sencillas pueden requerir solo cuatro señales (dos para datos y dos para handshake) mientras que otras pueden requerir solo señales de datos sin handshake. El listado completo de las señales definidas por el estándar se puede ver en [1].

Como se mencionó, rara vez se sigue al estándar RS-232 con precisión, debido principalmente a que muchas de las señales definidas no son necesarias para la mayoría de las aplicaciones. Muchas aplicaciones, como un módem, requieren solo nueve señales (dos señales de datos, seis señales de control y referencia). Otras aplicaciones pueden requerir solo cinco señales (dos para datos, dos para protocolo de enlace o handshake y referencia), mientras que otras pueden requerir solo señales de datos sin control de protocolo de enlace. Las señales necesarias para la conexión con un módem y los pines correspondientes del conector DB9 y DB25 se muestran en la tabla 1.2.

# pin DB-9 (DB-25)	Abrev.	Nombre completo	Sentido comun.
3 (2)	TD	Transmitted Data	DTE → DCE
2 (3)	RD	Received Data	DTE ← DCE
7 (4)	RTS	Request To Send	DTE → DCE
8 (5)	CTS	Clear To Send	DTE ← DCE
6 (6)	DSR	Data Set Ready	DTE ← DCE
5 (7)	SG	Signal Ground	DTE — DCE
1 (8)	DCD	Data Carrier Detect	DTE ← DCE
4 (20)	DTR	Data Terminal Ready	DTE → DCE
9 (22)	RI	Ring Indicator	DTE ← DCE

Tabla 1.2: Asignación de pines y sentidos de las señales del RS-232.

1.3.3. Características mecánicas

El tercer aspecto cubierto por estándar RS-232 hace referencia a la interfaz mecánica. En particular, el RS-232 especifica un conector de 25 pines. Este es el tamaño mínimo del conector que puede albergar todas las señales definidas en la parte funcional del estándar.

El conector para el equipo DTE tiene una carcasa hembra con pines de conexión macho, mientras que para el equipo DCE es macho para la carcasa y hembra para los pines. Aunque el RS-232 especifica un conector de 25 pines, el mismo rara vez se utiliza, debido al hecho de que la mayoría de las aplicaciones no requieren todas las señales definidas en el estándar. El conector más popular es el DB9 (9 pines) que se ilustra en la figura 1.7. Este conector resulta adecuado, por ejemplo, para transmitir y recibir las señales necesarias en aplicaciones con módems.

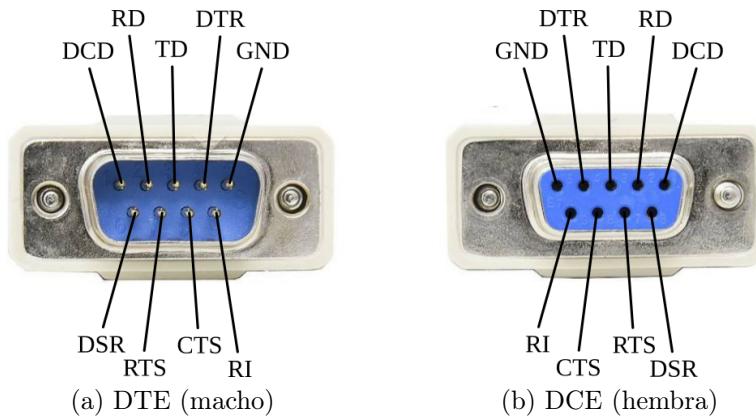


Figura 1.7: Conector DB9 y asignación de pines del RS-232.

1.3.4. Receptor/transmisor asíncrono universal, UART

Las señales necesarias para la comunicación en serie son generadas y recibidas por un circuito integrado (CI) conocido como UART (Universal Asynchronous Receiver/Transmitter). El CI que hace de driver de entrada/salida del RS-232 realiza la adaptación de nivel necesaria entre la interfaz CMOS/TTL y RS-232.

La UART realiza las tareas necesarias para la comunicación serial asíncrona la cual, por su naturaleza, es necesario que el sistema host genere los bits de inicio y parada para indicar al sistema periférico cuándo se inicia y termina la comunicación. Los bits de paridad pueden utilizarse también para garantizar que los datos enviados no se hayan dañado. La UART genera los bits de inicio, parada y paridad cuando transmite datos y puede detectar errores de comunicación al recibir datos. También funciona como intermediario entre la comunicación de a bytes y de a bits, convierte un byte de datos en un flujo de bits en serie en la transmisión y convierte un flujo de bits en serie en un byte de datos en la recepción.

La figura 1.8 muestra la conexión entre la UART, el circuito conversor de niveles de tensión y el conector típico para una comunicación serie con un módem, siguiendo el estándar RS-232.

1. Comunicación serie y estándar RS-232

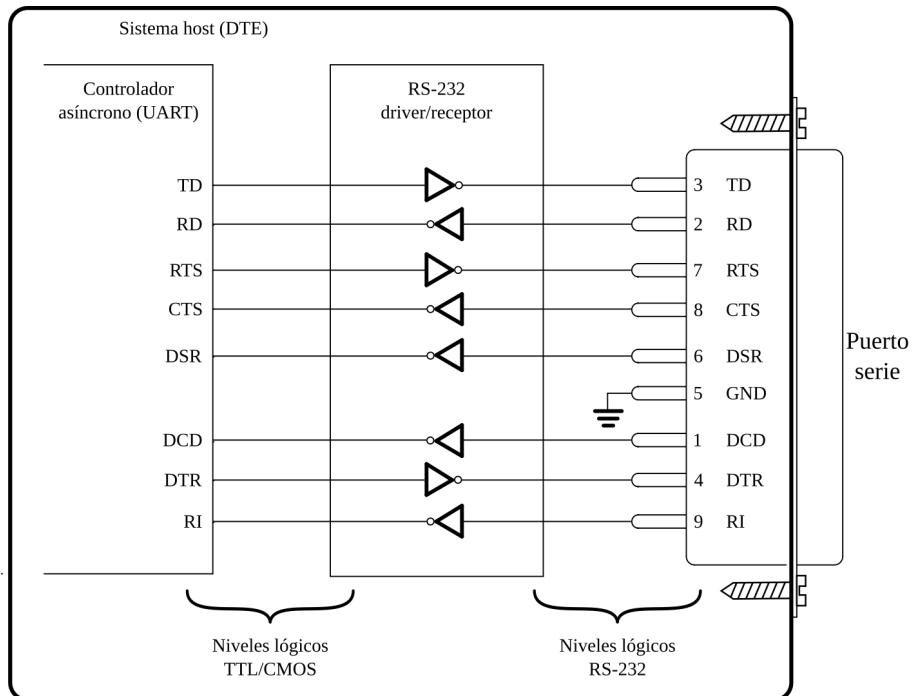


Figura 1.8: Aplicación típica del RS-232 en comunicación con módem.

1.4. Puerto serie en una PC

Previo a que el USB (Universal Serial Bus) se convirtiera en un estándar de facto como puerto de comunicación de las PC, existían diferentes puertos tales como el PS/2 (que toma su nombre de la serie de computadoras IBM Personal System/2), utilizado particularmente para teclado y mouse, y los antiguos puertos paralelos (IEEE-1284) y seriales (RS-232), utilizados en sus orígenes para la comunicación con impresoras y modems, respectivamente. El puerto paralelo y el serie eran los más adecuados para desarrollar circuitos electrónicos propios que tuvieran la capacidad de comunicarse con la PC.

Aún cuando es común hacer referencia a “puerto serie” como equivalente al estándar RS-232, este no es el único puerto serial usado para comunicar dispositivos electrónicos, como ya se mencionó anteriormente. En particular, en los sistemas embebidos están muy extendido el uso otro tipo de comunicación serie tales como SPI (Serial Peripheral Interface), I²C (Inter-Integrated Circuit), el ya mencionado USB, etc.

Si bien, el puerto de comunicación serie del estándar RS-232 no se encuentran en las PC o laptops actuales, es posible disponer del mismo mediante un adaptador o conversor USB-serie como los que se muestran en la figura 1.9. El adaptador de la figura 1.9a tiene un conector DB9 macho con todas las señales del estándar RS-232, mientras que los de las figuras 1.9b, 1.9c y 1.9d son conversores USB a niveles de tensión digitales comunes como 5V o 3.3V, los cuales tienen en general las señales TxD, RxD, GND y tensión de alimentación. Estos conversores también se conocen como conversores USB-TTL (Transistor-transistor Logic). El adaptador de la figura 1.9c tiene el circuito integrado PL2303 de la marca Prolific. Otros de los CIs de uso común en placas de conversión USB-TTL son el CP2101 y CP2102 de Silicon Lab. y el CH340. El adaptador de la figura 1.9d

utiliza el CI FT232 de la marca FTDI (Future Technology Devices International Ltd). La diferencia principal de esta placa con las demás es que pone a disposición en sus pines todas las señales de handshake del puerto serie.

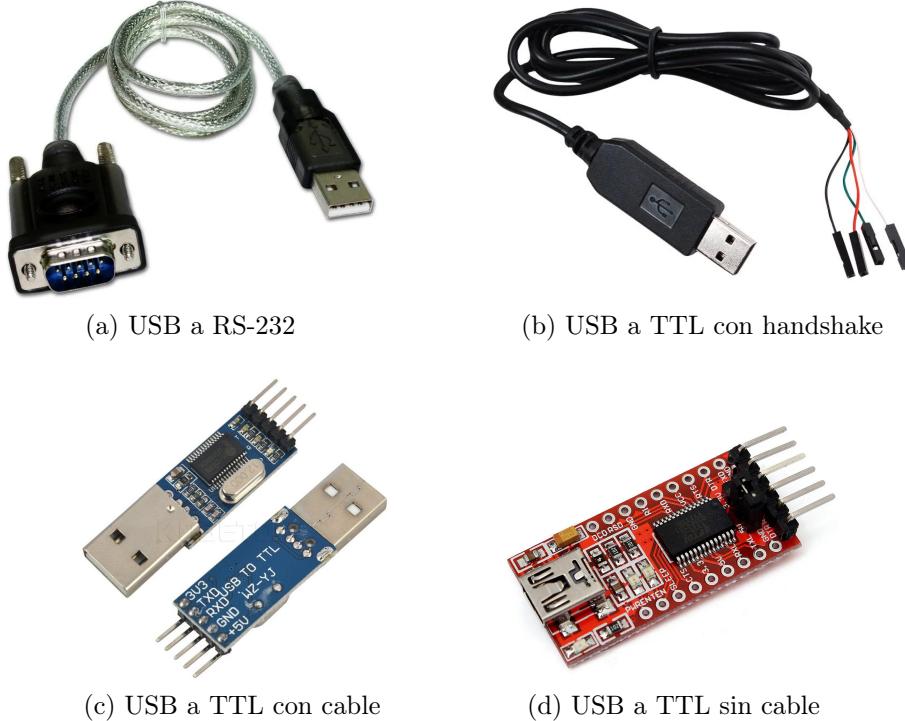


Figura 1.9: Adaptadores USB-serie.

1.4.1. Acceso a dispositivos

En GNU/Linux se tiene acceso a los dispositivos tales como los puertos series mediante archivos de dispositivos, los cuales se encuentran en el directorio `/dev`. Los archivos de dispositivos pueden ser de dos tipos: de caracteres y de bloques (indicados con la letra '`c`' y '`b`'). En el listado 1.1 se muestran tres archivos de dispositivo: dos de dispositivos de caracteres, `ttyUSBO` y `ttyACM0` de conversores USB-serie; y uno de dispositivo de bloques, el disco `sda`. El tipo de dispositivo está representado por la primer letra en la descripción detallada del archivo a la izquierda del listado, tal como la '`c`' en `crw-rw----`.

Listado 1.1: Archivos de dispositivos.

```
crw-rw---- 1 root dialout 188, 0 jul 20 14:02 /dev/ttyUSBO
crw-rw-rw- 1 root dialout 166, 0 jul 20 15:57 /dev/ttyACM0
brw-rw---- 1 root disk      8, 0 jul 20 14:02 /dev/sda
```

1. Comunicación serie y estándar RS-232

1.4.2. Dispositivos TTY

El nombre TTY, utilizado aún por razones históricas, significa TeleTYewriter (máquina de escribir remota). En los sistemas de tiempo compartidos anteriores a Unix se utilizaban terminales para interactuar con la computadora. Las terminales generaban los mensajes enviados desde un teletipo y la información recibida se imprimía localmente. En estos sistemas las terminales se conectaban de forma remota con la computadora mediante el cable adecuado y una UART.

En los sistemas operativos (SO) modernos como Unix o GNU/Linux se utilizan las terminales para interactuar con el mismo mediante dispositivos TTY. El funcionamiento interno de las terminales lo maneja directamente el kernel o núcleo del SO. Se puede acceder a las terminales de GNU/Linux presionando las teclas **Ctrl+Alt+Fx**, con x: 1,...,6. También existen programas emuladores de terminales tales como `gnome-terminal`, `konsole`, `xterm`, `terminator`, entre otras.

Un comando útil para ver y modificar la configuración de la terminal es `stty`. El listado 1.2 muestra la salida del comando `stty` en la terminal actual y el listado 1.3 de la terminal asociada al dispositivo `/dev/ttyUSB0`. La opción `-a` es para imprimir toda la información de configuración de la terminal, mientras que la opción `-F` sirve para indicarle el archivo de dispositivo. Se pueden apreciar los valores de algunos parámetros tales como: longitud de palabra, baudrate, control de flujo, paridad, etc.

Listado 1.2: Configuración de la terminal actual (`/dev/pts/4`).

```
$ stty
speed 38400 baud; line = 0;
-brkint -imaxbel iutf8
```

Listado 1.3: Configuración del dispositivo serial `/dev/ttyUSB0`.

```
$ stty -a -F /dev/ttyUSB0
speed 9600 baud; rows 0; columns 0; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <
        undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
        werase = ^W; lnext = ^V; discard = ^O; min = 1; time = 0;
-parenb -parodd -cmspar cs8 hupcl -cstopb cread clocal -crtsccts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -
        iuclc -ixany -imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel n10 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
        echoctl echoke -flusho -extproc
```

1.4.3. Comunicación entre dispositivos TTY

Es posible realizar una comunicación entre dos dispositivos TTY directamente desde la terminal de GNU/Linux. Para probar esta comunicación habría que disponer de dos adaptadores USB-serie conectados entre sí, o bien se puede utilizar la aplicación `socat`. `socat` permite generar dispositivos TTY que se comunican entre sí mediante sockets. Es como disponer de dos puertos seriales virtuales conectados entre sí. Por ejemplo, el comando

```
> socat pty,link=/tmp/ttyS0 pty,link=/tmp/ttyS1
```

crea los archivos de dispositivos TTY, /tmp/ttyS0 y /tmp/ttyS1. Se puede acceder a su configuración mediante > stty -a -F /tmp/ttyS0. Se puede modificar los parámetros de la comunicación agregando opciones al comando **socat**.

Una vez generados estos archivos y dejando abierta la terminal donde se encuentra ejecutado **socat**, se puede establecer una comunicación entre las dos terminales creadas. Por ejemplo, al ejecutar

```
> cat /tmp/ttyS1
```

en una terminal, y

```
> echo "Hola TTY" > /tmp/sttyS0
```

en otra, se estará enviando el mensaje "Hola TTY" desde ttyS0 a ttyS1.

También existen terminales con interfaz gráfica como por ejemplo **cuteCom** en el SO GNU/Linux. En la figura 1.10 se muestra una ventana de **cuteCom** que tiene abierto la terminal /tmp/ttyS1 con la configuración utilizada por **socat** (velocidad de transmisión de 38400 bps, etc). En la parte principal de la ventana se puede ver el mensaje recibido desde /tmp/ttyS0 generado por el comando **echo**.

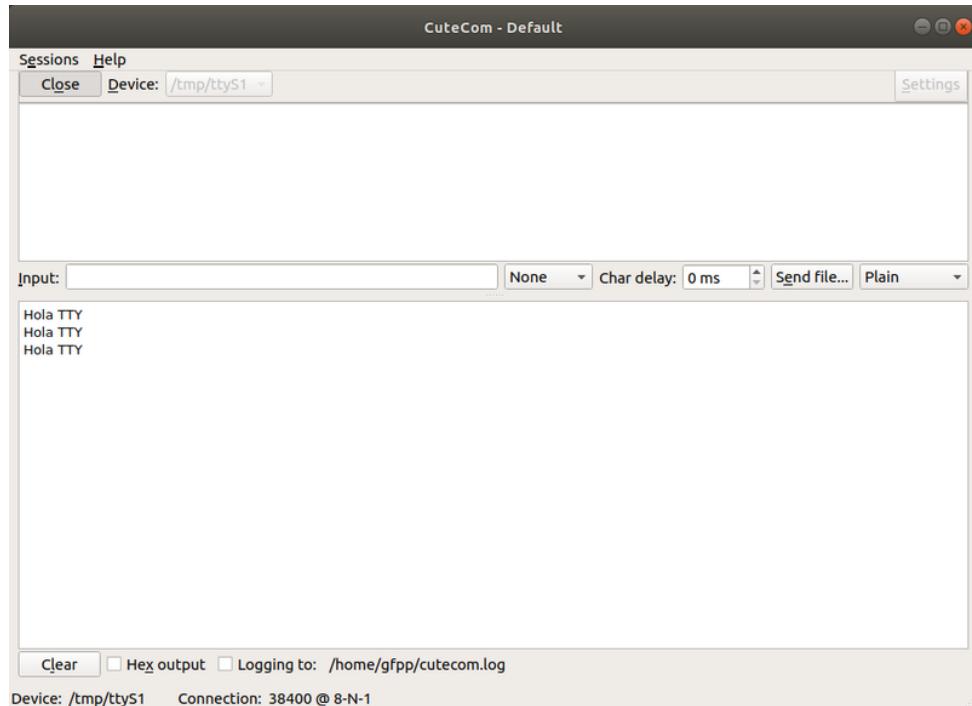


Figura 1.10: Terminal serial gráfica **cuteCom**.

Otros de los programas de terminal o emulador de terminal son: PuTTY que incluye una interfaz gráfica, y aplicaciones de consola como **screen**, **minicom**, entre otras.

1.5. Programación del puerto serie de la PC

Como se mencionó anteriormente, dentro del sistema operativo GNU/Linux el puerto serie se puede manejar utilizando el archivo de dispositivo de terminal (TTY) asociado. Para modificar la configuración de la terminal, en particular las opciones propias de la comunicación (trama de datos, velocidad de comunicación, control de flujo, etc.), se utilizan funciones de bibliotecas separadas de aquellas utilizadas para la lectura y escritura.

1.5.1. Programa ejemplo

En el listado 1.4 se muestra el código fuente para escribir un mensaje en el archivo asociado a un dispositivo serie, en este caso en `/tmp/ttyS0`. En este programa, primero se abre el archivo con la función `open()`, luego se escribe al archivo con la función `write()` y finalmente se cierra con la función `close()`. Los argumentos de la función `open()` son: el archivo (con su ruta absoluta) y las banderas (flags) que indican las propiedades al abrir el archivo. En el ejemplo la bandera `O_WRONLY` abre el archivo para solo escritura. En este ejemplo no se modifica la configuración de la terminal (utiliza la configuración por defecto).

Listado 1.4: Escritura en un archivo de dispositivo serie, `writetty.c`.

```
1 #include <stdio.h>
2 #include <unistd.h> /* Unix standard function definitions */
3 #include <fcntl.h> /* File control definitions */
4
5 int main(void)
6 {
7     int fd; /* File descriptor */
8
9     fd = open("/tmp/ttyS0", O_WRONLY);
10    if(fd == -1)
11    {
12        printf("ERROR: no se pudo abrir /tmp/ttyS0\n");
13        return -1;
14    }
15
16    write(fd, "Hola TTY\n", 9);
17    close(fd);
18    return 0;
19 }
```

El código fuente del listado 1.4 se puede compilar con

```
> gcc -Wall writetty.c -o writetty
```

lo que genera el archivo binario `writetty`. Este se puede ejecutar con

```
> ./writetty
```

Al ejecutar el programa se puede ver el mensaje enviado con `cutecom`, como en la figura 1.10.

1.5.2. Configuración de la terminal – estructura termios

`termios` es una interfaz especificada por el estándar POSIX (Portable Operating System Interface). La interfaz de terminal se controla fijando valores a un tipo estructura `termios` y utilizando un pequeño conjunto de funciones, ambos definidos en `termios.h`. Los valores que se pueden manipular para afectar el comportamiento de la terminal están agrupados en varios modos: entrada, salida, control, local y caracteres especiales. La definición mínima de estructura `termios` se muestra en el listado 1.5, donde los nombres de los miembros se corresponden con los tipos de parámetros ya mencionados.

Listado 1.5: Estructura `termios` definida en `termios.h`.

```

1 struct termios {
2     tcflag_t c_iflag;
3     tcflag_t c_oflag;
4     tcflag_t c_cflag;
5     tcflag_t c_lflag;
6     cc_t    c_cc[NCCS];
7 };

```

Se puede inicializar una estructura `termios` desde la configuración actual llamando a la función `tcgetattr()`, como

```
int tcgetattr(int fd, struct termios *term);
```

lo cual carga los valores actuales de configuración de la terminal a la estructura apuntada por el parámetro `term`. A partir de aquí se pueden cambiar los valores de los campos de la estructura y luego configurar la terminal con los valores modificados utilizando la función `tcsetattr()`.

```
int tcsetattr(int fd, int actions, const struct termios *term);
```

El parámetro `actions` controla cómo se aplican los cambios¹, pudiendo ser:

- `TCSANOW`: cambiar los valores inmediatamente.
- `TCSADRAIN`: cambiar los valores cuando la salida actual este completa.
- `TCSAFLUSH`: cambiar los valores cuando la salida actual este completa, pero descartando cualquier entrada disponible y que no ha sido leída aún por una llamada a `read()`.

1.5.3. Módulo para configuración del puerto serie

Para poder utilizar una función de configuración de puerto serie en diferentes aplicaciones es conveniente tener un módulo separado de dicha implementación². Este módulo está comprendido por el archivo de cabecera del listado 1.6 con la declaración de la función de configuración y el archivo de código fuente del listado 1.7 con la correspondiente implementación. En esta implementación, la función `termset` configura la comunicación

¹Ver página de manual: `man 3 tcsetattr`

²Un módulo de software es un conjunto de archivos `.h` y `.c`

1. Comunicación serie y estándar RS-232

serie con la trama más común de comunicación 8N1 (8 bits de datos, sin bit de paridad y 1 bit de stop), habilitado para lectura y sin control de flujo por hardware ni por software (XON/XOFF).

Listado 1.6: Archivo de cabecera del módulo, `termset.h`.

```
1 #ifndef TERMSET_H
2 #define TERMSET_H
3
4 #include <termios.h>
5
6 struct termios ttyold, ttynew;
7
8 /* termset function
9  * Parameters:
10 *   fd: file descriptor -device- (ex: /dev/ttyUSB0)
11 *   baudrate: communication speed (ex: 9600, 115200)
12 *   ttyold: current termios structure
13 *   ttynew: new termios structure
14 */
15 int termset(int fd, int baudrate, struct termios *ttyold,
16             struct termios *ttynew);
17
18 #endif
```

Listado 1.7: Archivo de código fuente del módulo, `termset.c`.

```

1 #include <stdio.h>
2 #include "termset.h"
3
4 int termset(int fd, int baudrate, struct termios *ttyold,
5     struct termios *ttynew)
6 {
7     switch(baudrate)
8     {
9         case 115200: baudrate = B115200;
10            break;
11        case 57600: baudrate = B57600;
12            break;
13        case 38400: baudrate = B38400;
14            break;
15        case 19200: baudrate = B19200;
16            break;
17        case 9600: baudrate = B9600;
18            break;
19        default:    baudrate = B115200;
20            break;
21    }
22
23    if(tcgetattr(fd, ttyold) != 0)
24    {
25        printf("ERROR: tcgetattr\n");
26        return -1;
27    }
28    ttynew = ttyold;
29
30    cfsetospeed(ttynew, baudrate);
31    cfsetispeed(ttynew, baudrate);
32
33    ttynew->c_cflag = (ttynew->c_cflag & ~CSIZE) | CS8;// 8 data bits (8)
34    ttynew->c_cflag &= ~(PARENB | PARODD);           // no parity (N)
35    ttynew->c_cflag &= ~CSTOPB;                      // 1 stop bit (1)
36
37    ttynew->c_cflag |= (CLOCAL | CREAD); // ignore modem status lines, and
38                                // enable reading
39    ttynew->c_cflag &= ~CRTSCTS;          // no flow control
40
41    ttynew->c_iflag &= ~IGNBRK;          // disable break processing
42    ttynew->c_iflag &= ~(IXON | IXOFF | IXANY); // shut off xon/xoff ctrl
43
44    ttynew->c_lflag = 0;                // no signaling chars, no echo,
45    ttynew->c_oflag = 0;                // no remapping, no delays
46    ttynew->c_cc[VMIN] = 0;             // read doesn't block
47    ttynew->c_cc[VTIME] = 100;          // read timeout
48
49 /*
50  * TCSANOW: Make the change immediately.
51  * TCSADRAIN: Make the change after all queued output has been written.
52  * TCSAFLUSH: This is like TCSADRAIN, but also discards any queued input.
53  */
54    if(tcsetattr(fd, TCSAFLUSH, ttynew) != 0)
55    {
56        printf("ERROR: tcsetattr\n");
57        return -1;
58    }
59
60    return 0;
61 }

```

1. Comunicación serie y estándar RS-232

Capítulo 2

Programación del microcontrolador ATmega328

2.1. Introducción

En este capítulo se describe la programación del microcontrolador ATmega328 incluido en varias placas Arduino, en particular la placa Arduino UNO, haciendo principal énfasis en la programación de los puertos digitales de entrada salida de propósito generales y de la comunicación serie asíncrona.

Se comienza con una breve descripción de la placa Arduino UNO para luego centrar la atención en el microcontrolador (μ C) que esta incluye, el ATmega328 [2]. Se describe las memorias presentes en este μ C y dos de los periféricos más importantes: i) los puertos digitales de entrada/salida y ii) el puerto de comunicación serie asíncrona (UART).

Desde el punto de vista de la programación primero se presenta cómo programar el μ C utilizando el IDE Arduino para luego pasar a la programación sin IDE, que será el método utilizado en el resto del documento. Finalmente, se muestra con varios ejemplos la programación de los periféricos de interés.

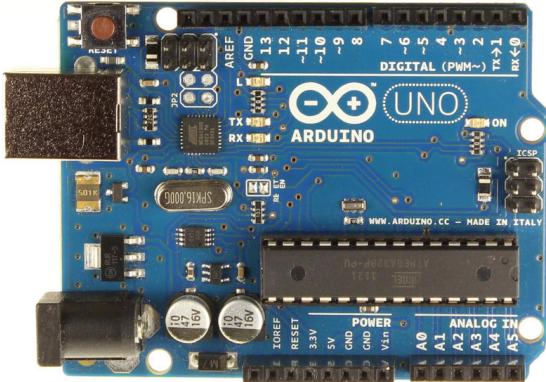
2.2. La placa Arduino UNO

La placa Arduino UNO es el primer modelo desarrollado por Arduino.cc [3] y es la placa más utilizada cuando se comienza a programar en el mundo Arduino [4]. La figura 2.1 muestra dos versiones de placa Arduino UNO más comunes: Arduino UNO Rev3 (original) en la figura 2.1a y Arduino UNO CH340 en la figura 2.1b. La principal diferencia entre estas dos versiones es el conversor USB-serie utilizado para la comunicación con la PC, utilizado principalmente para cargar el programa en el microcontrolador.

La placa Arduino UNO Rev3 [5] tiene un microcontrolador (μ C) marca Microchip[®] (antiguamente Atmel[®]) modelo ATmega328. Algunas de las características de esta placa son:

- Tensión de funcionamiento de 5V
- Tensión de entrada de 7 a 12V (de 6 a 20V como rango máximo)

2. Programación del microcontrolador ATmega328



(a) Arduino UNO Rev3



(b) Arduino UNO CH430

Figura 2.1: Diferentes versiones de placa Arduino UNO.

- 14 pines digitales de entrada/salida
 - 6 pines digitales con función PWM
 - 6 pines de entrada analógica

El conversor USB-serie de la placa Arduino UNO Rev3 (figura 2.1a) está basado en el μ C ATmega16U2 mientras que la placa Arduino UNO CH430 (figura 2.1b) utiliza el circuito integrado (CI) CH430.

La figura 2.2 muestra el pinout de la placa Arduino UNO.

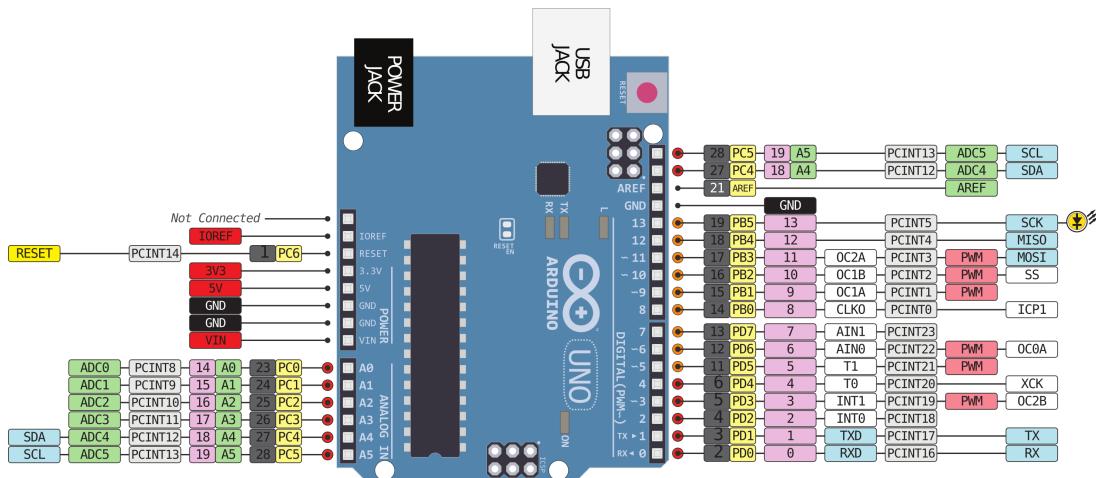


Figura 2.2: Pinout de la placa Arduino UNO.

2.3. El microcontrolador ATmega328

El microcontrolador ATmega328 es un μ C de 8-bits de arquitectura AVR RISC mejorado. Algunas de las características de este μ C son:

- Arquitectura RISC avanzada:
 - 131 instrucciones (la mayoría de las cuales se ejecutan en un ciclo de reloj)
 - 32 registros de propósitos generales de 8-bits
 - Máximo de 20MIPS (million instructions per second) @ 20MHz de frecuencia de reloj
 - Multiplicación por hardware
- Memoria:
 - 32KB de memoria FLASH
 - 2KB de memoria SRAM (Static Random Access Memory)
 - 1KB de memoria EEPROM (Electrically Erasable Programmable Read-Only Memory)
- Periféricos:
 - 23 entrada-salida de propósito general (GPIO: General Purpose Input-Output)
 - 6 canales de PWM (Pulse Width Modulation)
 - 6/8 canales de ADC (Analog to Digital Converter) de 10-bits
 - Puerto serial programable (USART: Universal Synchronous/Asynchronous Receiver/Transmitter)
 - Interfaz serial SPI (Serial Peripheral Interface) maestro-esclavo
 - Interfaz serial de 2-cables (compatible con el I²C de Philips)
 - Interrupción externa por cambio de nivel en entrada digital

La figura 2.3 muestra dos de los encapsulados disponibles para el μ C ATmega328: PDIP (Plastic Dual-In Line Package) de 28 pines en la figura 2.3a y TQFP (Thin Quad Flat Package) de 32 pines en la figura 2.3b.

Como se observa en la figura 2.3, la mayoría de los pines del μ C pueden multiplexar diferentes funciones de los periféricos disponibles. Por ejemplo: el pin nro. 2 del encapsulado PDIP puede actuar como bit 0 del Puerto D o como señal RxD de la UART o puerto serie.

2.3.1. Memoria

El μ C ATmega328 tiene dos espacios de memorias diferentes: una es la memoria de programa y la otra la memoria de datos. Además, dispone de una memoria EEPROM para datos. Las tres memorias se organizan en espacios lineales y regulares.

La memoria Flash, para almacenar el programa en el μ C ATmega328, es de $16K \times 16$ -bits haciendo un total de 32KBytes (32×8 -bits), y tiene una capacidad de escritura y borrado de al menos 10.000 ciclos. El contador de programa (PC: Program Counter) es de 14-bits lo que permite direccionar 16K ubicaciones de la memoria de programa. La

2. Programación del microcontrolador ATmega328

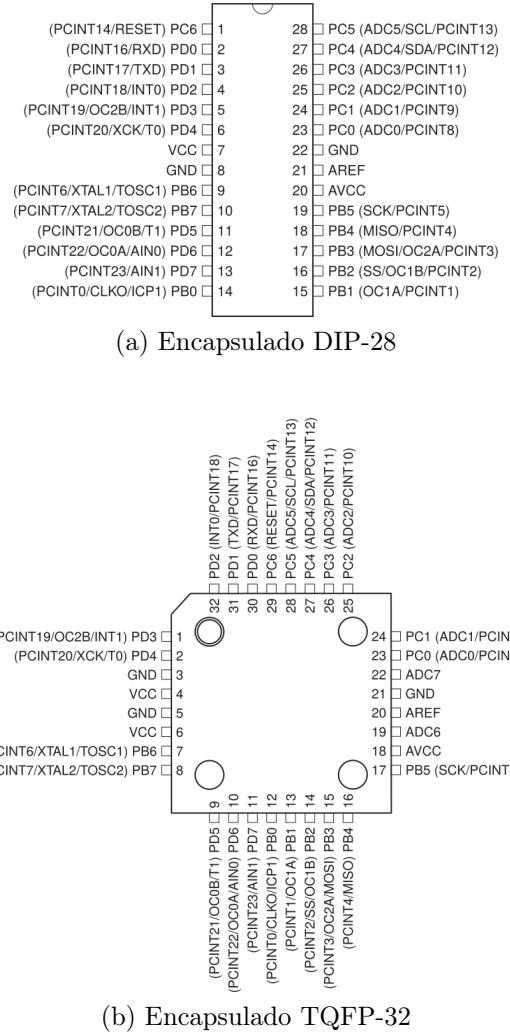


Figura 2.3: Encapsulados más comunes del μ C ATmega328.

figura 2.4 muestra la organización de la memoria de programa Flash (mapa de memoria) del μ C ATmega328.

La memoria RAM se organiza en diferentes secciones. Las primeras direcciones están reservadas para acceder a los registros de propósito general y los de entrada-salida, y las siguientes para almacenar las variables del programa. Esta última sección tiene un total de 2KBytes ($2K \times 8\text{-bits}$). La figura 2.5 muestra la organización de la memoria de datos RAM (mapa de memoria) del μ C ATmega328.

2.3.2. Puerto digital de entrada/salida

Como se mencionó, la mayoría de los pines del μ C ATmega328 pueden multiplexar diferentes funciones de los distintos periféricos, siendo una de ellas las de actuar como entrada/salida digital de propósito general (GPIO: General Purpose Input/Output). En esta sección se muestra cómo manejar los puertos de entrada/salida digitales del μ C ATmega328.

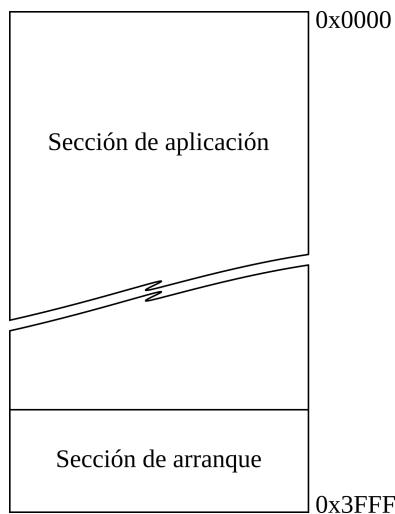


Figura 2.4: Memoria de programa (Flash) del μ C ATmega328.

32 registros	0x0000 - 0x001F
64 registros de E/S	0x0020 - 0x005F
	0x0100
Memoria RAM gral (2048 x 8-bits)	0x08FF

Figura 2.5: Memoria de datos (RAM) del μ C ATmega328.

Registros

Para el manejo de los puertos digitales de entrada/salida se utilizan 3 registros: DDRx, PORTx y PINx donde la x tiene que ser reemplazada por un puerto en particular, o sea B, C o D para el μ C ATmega328. Por ejemplo, para el manejo del puerto D se utilizan los registros DDRD, PORTD y PIND. Las funciones de estos registros son:

- DDRx: registro de dirección (Data Direction Register) del puerto x (B, C o D).
- PORTx: registro de datos (Data Register) del puerto x (B, C o D).
- PINx: registro de lectura (Input Pin Register) del puerto x (B, C o D)¹.

Los registros de manejo de los puertos digitales de entrada/salida se encuentran mapeados en la memoria RAM de μ C (ver figura 2.5), o sea que tienen direcciones de memorias fijas. La tabla 2.1 muestra las direcciones de memoria de los registros para el manejo de los puertos B, C y D del μ C ATmega328.

Algunas de las características eléctricas de los puertos digitales de entrada/salida son: los búfer de las salidas digitales de todos los pines tienen características de corriente simétrica lo que significa que pueden drenar o suministrar la misma corriente eléctrica;

¹Escribir un uno lógico en PINxn alterna el valor de PORTxn, independientemente del valor de DDRxn.

2. Programación del microcontrolador ATmega328

	PINx	DDRx	PORTx
Puerto B	0x23	0x24	0x25
Puerto C	0x26	0x27	0x28
Puerto D	0x29	0x2A	0x2B

Tabla 2.1: Dirección de los registros de manejo de GPIO.

y, configurados como salida, los pines tienen suficiente capacidad de corriente como para manejar displays LED de forma directa.

Cabe mencionar que el único puerto disponible en su totalidad (8 bits) en la placa Arduino UNO es el Puerto D (ver figura 2.2). Si bien el Puerto B también está disponible en su totalidad tanto en la versión DIP como SMD del μ C, este no se encuentra disponible en los pines de la placa debido a que están multiplexados con los pines para el oscilador a cristal de cuarzo (PB6 y PB7).

La figura 2.6 muestra los registros para el manejo de los puertos. El μ C ATmega328 dispone de un conjunto de estos 3 registros (DDRx, PORTx y PINx) para cada uno de los puertos disponibles. En la figura se observa el nombre de cada registro y de los bits que los componen, si acepta la operación de lectura y/o escritura y el estado inicial. La x minúscula representa la letra de numeración para el puerto y la n minúscula el número de bit.

Bit	7	6	5	4	3	2	1	0	DDRx
Read/Write	DDx7	DDx6	DDx5	DDx4	DDx3	DDx2	DDx1	DDx0	
Initial value	R/W 0								

Bit	7	6	5	4	3	2	1	0	PORTx
Read/Write	PORTx7	PORTx6	PORTx5	PORTx4	PORTx3	PORTx2	PORTx1	PORTx0	
Initial value	R/W 0								

Bit	7	6	5	4	3	2	1	0	PINx
Read/Write	PINx7	PINx6	PINx5	PINx4	PINx3	PINx2	PINx1	PINx0	
Initial value	R/W N/A								

Figura 2.6: Registros para el manejo de puertos digitales de entrada/salida.

Los bits DDxn del registro DDx seleccionan si el pin correspondiente actuará como entrada o salida. Si se escribe un 1 lógico a DDxn el pin Px_n se configura como salida. Si se escribe un 0 lógico a DDxn, el pin Px_n se configura como entrada.

Si se escribe un 1 lógico en PORTxn cuando el pin está configurado como salida, el pin del puerto se pondrá en alto (uno). Si se escribe un 0 lógico en PORTxn cuando el pin está configurado como salida, el pin del puerto se pondrá en bajo (cero). Si se escribe un 1 lógico a PORTxn cuando el pin está configurado como entrada, se activará la resistencia de pull-up. Para desactivar (desconectar) la resistencia de pull-up se debe escribir un 0 lógico a PORTxn o bien configurar el pin como salida.

Se puede leer el valor del bit de un puerto con PINxn, independientemente del valor que tenga el bit DDxn correspondiente. Al escribir un 1 lógico en PINxn cambiará el valor (toggle) de PORTxn correspondiente, independientemente del valor de DDxn.

2.3.3. Puerto serie USART

El microcontrolador ATmega328 posee un periférico denominado USART, por las siglas en inglés de Universal Synchronous and Asynchronous Receiver and Transmitter que permite la comunicación serie con otro dispositivo, como por ejemplo una PC. La USART del ATmega328 se denomina USART0 dado que otros modelos de μ C pueden tener más de una USART (USART0, USART1, etc.).

En esta sección se describe la USART0 del μ C ATmega328 para la comunicación serie asíncrona. Para este tipo de comunicación se utiliza un pin digital para la salida de los datos TXD y un pin digital para la entrada de los datos RXD (ver figura 2.3). Otro tipo de comunicación soportada por la USART del μ C ATmega328 es el modo SPI (Serial Peripheral Interface) Maestro.

Algunas de las características de la USART del μ C ATmega328 son:

- Permite operación Full-duplex (tiene registros independientes de transmisión y recepción)
- Permite operación síncrona (modo SPI) y asíncrona
- Soporta tramas seriales con 5, 6, 7, 8 y 9 bits de datos, y 1 o 2 bits de parada
- Generación de paridad impar o par y verificación de paridad por hardware

El término “Duplex” en telecomunicaciones se utiliza para definir un sistema que permite una comunicación bidireccional (en ambos sentidos), y Full-duplex significa que la comunicación bidireccional puede darse de forma simultánea.

Trama de comunicación

La trama (frame) de comunicación de la USART consiste en una palabra de entre 5 y 9 bits de datos que se envían o reciben de forma serializada. La trama se muestra de forma esquemática en la figura 2.7.

Todas las tramas comienzan con un bit de inicio seguido por el bit de datos menos significativo (LSb, Least Significant bit), luego se envían los siguientes bits de datos hasta un total de nueve, finalizando con el bit más significativo (MSb, Most Significant bit). Si está habilitado, se inserta un bit de paridad luego de los bits de datos y antes del bit de parada. Cuando se completa una transmisión puede seguir de forma inmediata una nueva trama o la línea de comunicación se puede fijar a estado ocioso (idle).

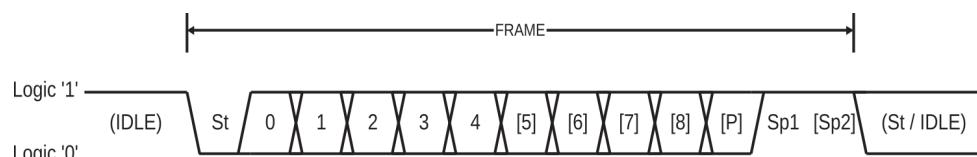


Figura 2.7: Trama (frame) de comunicación serie.

La nomenclatura utilizada en la figura 2.7 es:

- **St:** Bit de inicio de trama (nivel bajo)

2. Programación del microcontrolador ATmega328

- **(n):** Bits de datos (0 a 8)
- **P:** Bit de paridad, que puede ser impar o par
- **Sp:** Bit de parada (nivel alto)
- **IDLE:** No hay transferencia de datos sobre las líneas de comunicación (RXD o TXD) (nivel alto)

Diagrama en bloques

El diagrama en bloques de la USART del μ C ATmega328 se muestra en la figura 2.8. Como se observa, consiste de 3 bloques principales: i) el generador de reloj (clock), ii) el bloque de transmisión y iii) el de recepción. También pueden observarse los pines TxDn, RxDn y el pin XCKn el cual se utiliza únicamente cuando se activa el modo de transmisión síncrona. Los registros accesibles desde la CPU (remarcados en negrita) son:

- **UDR_n:** Registros búfer de transmisión y recepción de datos (USART Data Registers)
- **UBRR_{n[H:L]}:** Registros de velocidad (baud-rate) de transmisión (USART Baud Rate Registers)
- **UCSR_{nA}, UCSR_{nB}, UCSR_{nC}:** Registros de estado y control (USART Control and Status Registers)

La lógica para la generación de reloj genera los pulsos de reloj (clock) base para el transmisor y el receptor. La USART soporta 4 modos de operación: i) asíncrono normal, ii) asíncrono de doble velocidad, iii) síncrono maestro y iv) síncrono esclavo.

El registro Baud Rate de la USART (UBRR_n) en conjunto con un contador descendente (down-counter) funcionan como un prescaler o generador de baud-rate programable. El contador descendente, corriendo a la velocidad de reloj del sistema (f_{osc}), se carga con el valor del registro UBRR_n cada vez que alcanza el valor cero o cuando se escribe el registro UBRR_{nL} y genera un pulso de reloj. El reloj del generador de baud-rate tiene entonces una frecuencia $f_{osc}/(UBRR_n + 1)$ la cual se divide luego por 2, 8 y 16 dependiendo del modo.

En el modo asíncrono normal la tasa de transmisión de datos (BR) es

$$BR = \frac{f_{osc}}{16 \cdot (UBRR_n + 1)}, \quad (2.1)$$

por lo tanto

$$UBRR_n = \frac{f_{osc}}{16 \cdot BR} - 1. \quad (2.2)$$

donde UBRR_n es el valor del registro de velocidad.

La tabla 2.2 muestra las velocidades de comunicación más utilizadas y el valor a cargar en el registro UBRR_n cuando el μ C ATmega328 tiene un oscilador de cristal de cuarzo

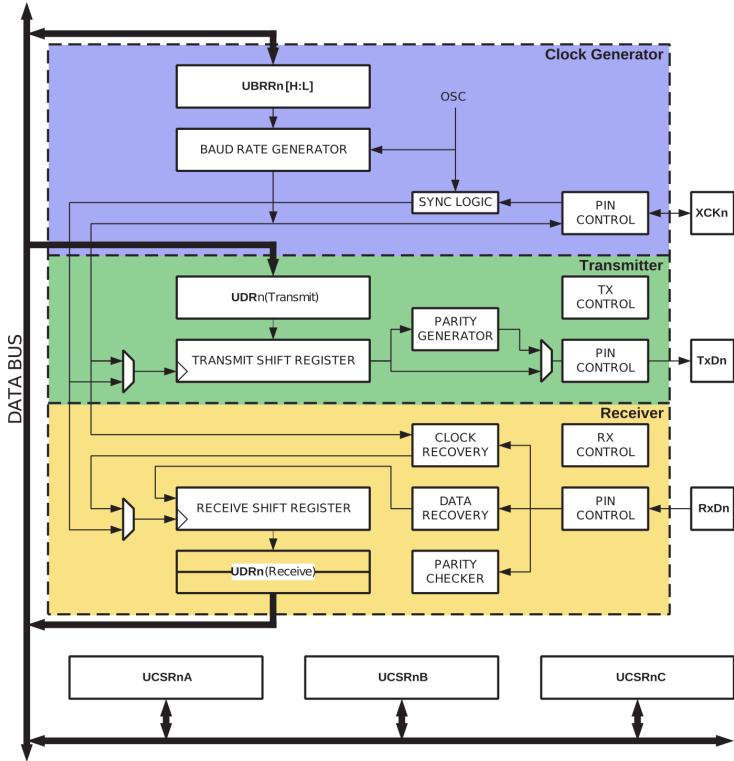


Figura 2.8: Diagrama en bloques del módulo USART.

Baud Rate (bps)	2400	4800	9600	14.4K	19.2K	28.8K	38.4K	57.6K	76.8K	115.2K
UBRRn	416	207	103	68	51	34	25	16	12	8
Error (%)	-0.1	0.2	0.2	0.6	0.2	-0.8	0.2	2.1	0.2	-3.5

 Tabla 2.2: Velocidades de comunicación y valor del registro UBRRn para $f_{osc} = 16MHz$.

de 16MHz como es el caso de la placa Arduino UNO. Se muestra también el error en la velocidad dado el valor del divisor entero.

Por otro lado, el bloque transmisor consiste en un búfer de escritura, un registro de serialización o desplazamiento (Shift Register), el generador de paridad y la lógica de control para los diferentes formatos de trama; mientras que el receptor consiste de la lógica de control y verificación de paridad, el registro des-serializador y el búfer de recepción de dos niveles. Los errores que pueden producirse en la recepción de datos son: error de trama, error de desbordamiento (OverRun) y error de paridad.

Para poder utilizar la USART en una comunicación es necesario su inicialización. En general, el proceso de inicialización consiste en configurar el baud-rate, el formato de la trama y habilitar el transmisor y/o el receptor. Este proceso se realiza utilizando los registros antes mencionados.

La configuración de la USART0 del μ C ATmega328 para una comunicación asíncrona se realiza mediante los siguientes pasos:

1. Configuración del baud-rate: se realiza escribiendo los registros **UBRR0[H:L]**

2. Programación del microcontrolador ATmega328

2. El formato de la trama de comunicación se configura utilizando el registro **UCSR0C**
3. La habilitación de la transmisión y recepción se realiza a través del registro **UCSR0B**

Registros

Para el manejo de la USART0 del μ C ATmega328 se utilizan 5 registros: UDR0, UCSROA, UCSR0B, UCSR0C y UBRRO. Estos registros están mapeados en la memoria RAM del μ C (ver figura 2.5), o sea que tienen direcciones de memorias fijas. La figura 2.9 muestra los registros para el manejo de la USART0, los bits que los componen y las direcciones de memoria donde están mapeados. A continuación se describen las funciones de los más relevantes.

Bit	7	6	5	4	3	2	1	0	
0xC6	RXB[7:0]								UDR0 (Read)
	TXB[7:0]								UDR0 (Write)
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	
0xC0	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0	UCSR0A
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial value	0	0	1	0	0	0	0	0	
0xC1	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80	UCSR0B
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial value	0	0	0	0	0	0	0	0	
0xC2	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0	UCSR0C
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial value	0	0	0	0	0	1	1	0	
0xC5	-	-	-	-	UBRR0[11:8]				UBRR0H
0xC4	UBRR0[7:0]								UBRR0L
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

Figura 2.9: Registros para el manejo de la USART0 del μ C ATmega328.

Registro de datos de la USART0

Los registros búfer de transmisión y de recepción de datos de la USART comparten la misma dirección de entrada/salida como registro UDR0 (USART Data Register). Cuando se realiza una escritura en la dirección del registro UDR0, los datos se almacenan en el registro búfer de transmisión (TXB). Cuando se lee la dirección de memoria del registro UDR0 se obtiene el valor del registro búfer de recepción de datos (RXB). Los bits más altos no son utilizados en el caso de tramas de 5-, 6- y 7-bits cuando se escribe y en la lectura son puestos a cero por el receptor.

El búfer de transmisión solo se puede escribir cuando el bit UDRE0 del registro UCSROA este a 1, en caso contrario los datos escritos serán ignorados por el transmisor de la USART. Cuando los datos se escriben en el búfer de transmisión y este está habilitado, el transmisor cargará el dato en el registro serializador de transmisión (Transmit Shift

Register) cuando este este vacío. Luego, los datos se transmitirán en serie por el pin TxD0. En cuanto a la recepción, el búfer de recepción consta de una FIFO de dos niveles, cuyo estado cambia cada vez que se acceda al búfer de recepción.

Registro de control y estado A de la USART0

Las funciones de los bits más utilizados del registro de control y estado A de la USART0 son:

- Bit 7 – RXC0 (USART Receive Complete / Recepción completa): este bit actúa como bandera (flag) para indicar que hay un dato no leído en el búfer de recepción y se borra cuando el búfer de recepción está vacío.
- Bit 6 – TXC0 (USART Transmit Complete / Transmisión completa): este bit de bandera se pone a 1 cuando la trama completa en el registro de desplazamiento de transmisión ha sido serializado y no hay datos nuevos en el búfer de transmisión (UDR0).
- Bit 5 – UDRE0 (USART Data Register Empty / Registro de datos vacío): este flag indica si el búfer de transmisión (UDR0) está listo para recibir un dato nuevo. Si está a 1 el búfer está vacío y listo para ser escrito.
- Bit 4 – FEO (Frame Error / Error de trama):
- Bit 3 – DOR0 (Data OverRun / Desbordamiento de datos):
- Bit 2 – UPE0 (USART Parity Error / Error de paridad):

Registro de control y estado B de la USART0

Las funciones de los bits más utilizados del registro de control y estado B de la USART0 son:

- Bit 4 – RXENO (Receiver Enable / Receptor habilitado): al escribir un 1 se habilita la recepción de datos y se configura el pin RxD0.
- Bit 3 – TXENO (Transmitter Enable / Transmisor habilitado): al escribir un 1 se habilita la transmisión de datos y se configura el pin TxD0.
- Bit 2 – UCSZ02 (Character Size / Tamaño del carácter): este bit, en combinación con los bits UCSZ01:0, configuran la cantidad de bits de datos (Character SiZe) de la trama de transmisión y recepción.

Registro de control y estado C de la USART0

Las funciones de los bits más utilizados del registro de control y estado C de la USART0 son:

- Bits 7:6 – UMSEL01:0 (USART Mode Select / Selección del modo de la USART): estos bits seleccionan el modo de operación de la USART0, donde para el modo asíncrono los valores deben ser 00.

2. Programación del microcontrolador ATmega328

- Bits 5:4 – UPM01:0 (Parity Mode / Modo de paridad): fijan el tipo de paridad utilizada. Estos son: 00, paridad desactivada; 01, reservado; 10, paridad par; y 11, paridad impar.
- Bit 3 – USBS0 (Stop Bit Select / Selección de bit de stop): seleccionan la cantidad de bits de stop agregados en la transmisión (el receptor ignora esta configuración). Las opciones son 0: 1-bit de parada, o 1: 2-bits de parada.
- Bit 2:1 – UCSZ01:0 (Character Size / Tamaño de carácter): estos bits, en conjunto con el bit UCSZ02 en UCSR0B, configuran la cantidad de bits de datos (Character SiZe) utilizada en la trama de transmisión y recepción. La tabla 2.3 muestra las opciones de configuración.

UCSZ02	UCSZ01	UCSZ00	Tamaño de carácter
0	0	0	5 bits
0	0	1	6 bits
0	1	0	7 bits
0	1	1	8 bits
1	0	0	Reservado
1	0	1	Reservado
1	1	0	Reservado
1	1	1	9 bits

Tabla 2.3: Configuración de los bits UCSZ0.

Registros de Baud Rate

Los bits del registro de baud rate son:

- Bits 15:12: estos bits están reservados para uso futuro.
- Bits 10:0 – UBRR[11:0] (USART Baud Rate): estos 12 bits configuran la velocidad de transmisión (baud rate) de la USART. El registro UBRR0H contiene los 4 bits más significativos y el registro UBRR0L los 8 bits menos significativos del baud rate. Al escribir el registro UBRR0L actualiza el prescaler del baud rate de forma inmediata.

2.4. Programación del ATmega328

El microcontrolador ATmega328 de la placa Arduino UNO puede programarse ya sea utilizando o no el IDE Arduino. Ambas opciones se muestran a continuación en conjunto con sendos ejemplos de programas *Hola mundo* para sistemas embebidos, que consiste en hacer parpadear un LED (Blink) conectado a un pin digital (GPIO) configurado como salida.

Los diferentes archivos de código fuente mostrados como ejemplo a continuación se pueden obtener de [6]. Si no se dispone de una placa Arduino UNO se puede utilizar el simulador online TinkerCAD® [7].

2.4.1. Programación con el IDE Arduino

Los programas escritos utilizando el Entorno de Desarrollo Integrado o IDE por sus siglas en inglés de Integrated Development Environment se llaman sketch, que podría traducirse como “boceto” [4]. La idea de “sketch” en el contexto de la programación proviene de Processing [8]. Processing es un lenguaje de programación y entorno de desarrollo integrado de código abierto basado en Java para la enseñanza y producción de proyectos multimedia e interactivos, sobre el cual se basó el desarrollo del IDE Arduino. Esta idea se extendió mediante la placa de desarrollo Wiring dentro del contexto de la creación de prototipos de electrónica o “sketching” con hardware [4].

La ventana o interfaz gráfica de usuario (GUI, Graphical User Interface) del IDE Arduino se muestra en la figura 2.10. Los sketches se escriben en el área de edición de texto del IDE y se guardan con extensión .ino.

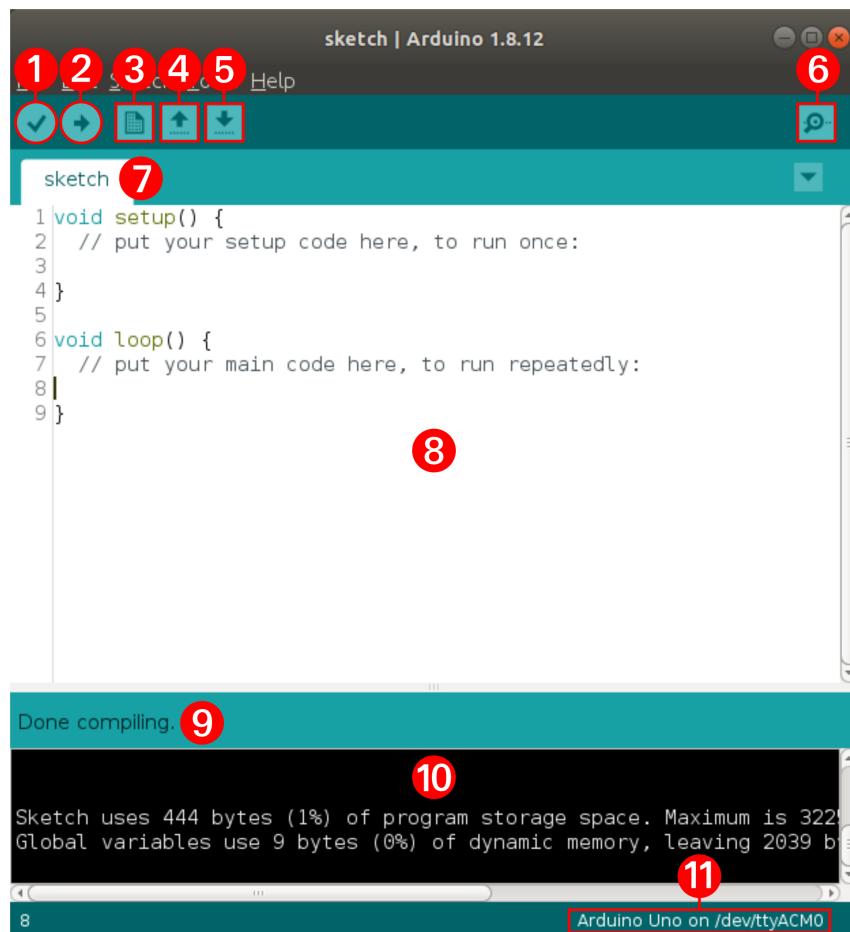


Figura 2.10: Entorno de desarrollo integrado (IDE) Arduino.

Las diferentes partes de la GUI del IDE son:

1. **Construcción (Verify)**: realiza la construcción de los archivos del proyecto
2. **Grabar (Upload)**: carga el programa construido en la memoria del μ C

2. Programación del microcontrolador ATmega328

3. **Nuevo (New)**: generar un archivo nuevo (sketch)
4. **Abrir (Open)**: abrir un archivo de proyecto (sketch)
5. **Guardar (Save)**: guarda el archivo actual (sketch)
6. **Monitor Serial (Serial Monitor)**: abre la ventana del monitor serial
7. **Nombre del sketch**: muestra el nombre del sketch actual
8. **Editor**: es el área para la edición del código fuente
9. **Área de mensajes**: indica si hubo un error o no en la construcción
10. **Consola**: muestra los mensajes de construcción y grabación
11. **Modelo de placa y puerto serie**: muestra el modelo de placa y puerto serie configurados

Para la instalación del IDE Arduino en Ubuntu hay que descargar el archivo `arduino-x.y.z-linux64.tar.xz` (para sistemas de 64-bits o el archivo correspondiente para 32-bits) [9], donde `x.y.z` hace referencia a la versión del IDE, descomprimir el archivo y ejecutar el script de instalación `install.sh`. El IDE de Arduino incluye:

1. La biblioteca de programación [10]
2. La herramienta de construcción de sketch `arduino-builder`, la cual realiza la conversión de archivos `.ino` a archivos en lenguaje C++ con extensión `.cpp`
3. El conjunto de herramientas para la construcción de programas (Toolchain) basado en GCC

Desde el menú **Tools** del IDE se puede seleccionar la placa Arduino particular sobre la que se trabajará con la opción **Board** y el puerto serie por el cual se grabará el programa en la memoria del μ C en la opción **Port**. En la figura 2.11 se observa el menú **Tools** desplegado, donde para este caso particular la placa seleccionada es la “Arduino UNO” y el puerto serie es `/dev/ttyACM0`.

El monitor serial (Serial Monitor) es una herramienta que incluye el IDE que se abre como ventana separada y actúa como una terminal de comunicación serial para recibir y enviar datos desde y hacia el μ C ATmega328. La figura 2.12 muestra la ventana del monitor serial del IDE Arduino.

Las diferentes partes de la GUI del monitor serial son:

1. Campo de edición para escribir el mensaje a enviar al μ C
2. Botón para enviar el mensaje del campo de edición
3. Área donde se muestran los mensajes recibidos desde el μ C
4. Permite activar/desactivar el desplazamiento (scroll) automática del área de mensajes recibidos

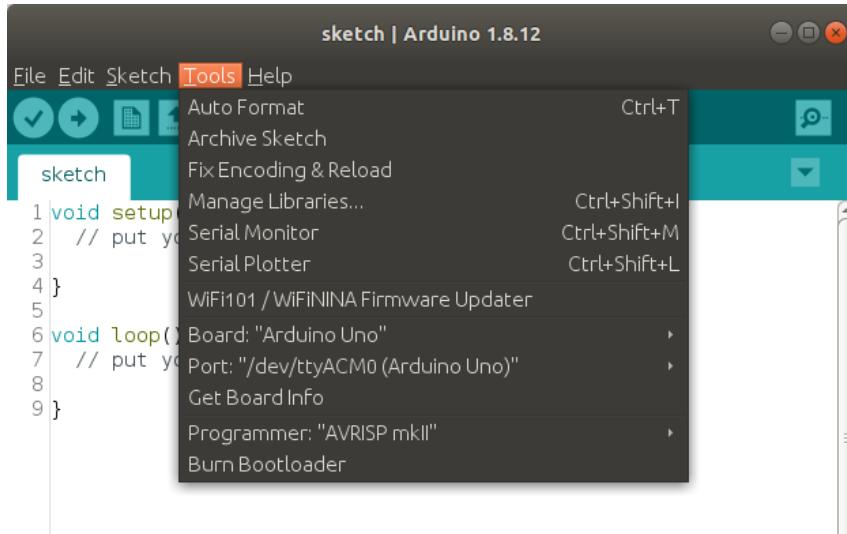


Figura 2.11: Menú “Tools” del IDE Arduino.

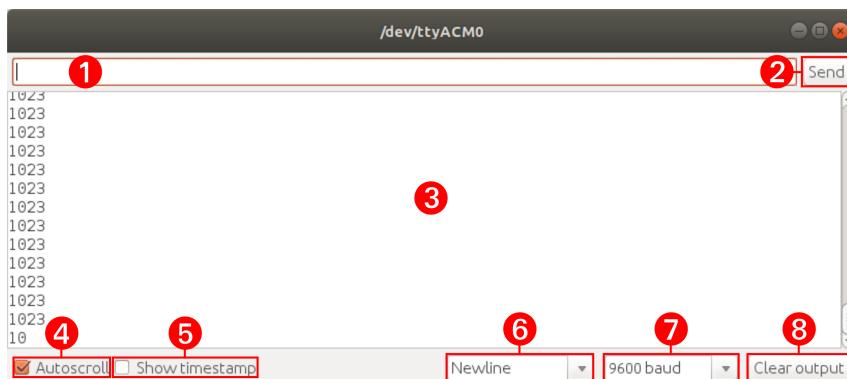


Figura 2.12: Monitor serial del IDE Arduino.

5. Agrega una marca de tiempo (timestamp) en los mensajes recibidos
6. Menú desplegable para definir el fin del mensaje enviado. Puede ser: No line end (sin final), New line (caracter de nueva línea), Carriage return (caracter de retorno de carro), Both NL & CR (caracteres de nueva línea y retorno de carro)
7. Menú desplegable para seleccionar la velocidad de comunicación (baud)
8. Botón para borrar el área de mensajes recibidos

En la ventana del monitor serial mostrada en la figura 2.12 se observa que se está recibiendo el valor 1023 (cadena "1023"), está activado el Autoscroll y que la velocidad de comunicación está configurada en 9600 baudios o bits por segundo (bps).

Programación de sketch

Como se mencionó, los programas escritos con el IDE Arduino se llaman sketches y se guardan con la extensión .ino. La programación del sketch consiste en escribir el cuerpo

2. Programación del microcontrolador ATmega328

de dos funciones:

- **setup()**: se ejecuta cuando se enciende o reinicia (reset) el microcontrolador
- **loop()**: se ejecuta de forma constante en un bucle infinito

En el menú **File->Examples** se encuentran varios sketches de ejemplos que pueden servir como punto de partida para conocer las funciones de la biblioteca Arduino. Algunos de los ejemplos interesantes para estudiar son: **DigitalReadSerial**, **AnalogReadSerial**, **Fade** y **BlinkWithoutDelay**, entre otros.

El listado 2.1 se muestra un sketch para encender y apagar un LED cada 1 seg. En la función **setup()** se configura como salida el pin digital sobre el cual se encuentra conectado el LED llamando a la función **pinMode()**. Luego, en la función **loop()** se enciende y apaga el LED con la función **digitalWrite()** cada un segundo. El retardo de tiempo (delay) se realiza con la función bloqueante **delay()** cuyo argumento determina el tiempo de retardo expresado en milisegundos.

Listado 2.1: Código fuente de sketch Arduino **Blink.ino**.

```
1 #define LED LED_BUILTIN
2
3 // La función 'setup' se ejecuta una única vez al encender la placa
4 // o presionar reset
5 void setup() {
6     // Inicializa el pin digital como salida
7     pinMode(LED, OUTPUT);
8 }
9
10 // La función 'loop' corre indefinidamente una y otra vez.
11 void loop() {
12     digitalWrite(LED, HIGH);    // Enciende el LED
13     delay(1000);              // Espera 1 seg.
14     digitalWrite(LED, LOW);   // Apaga el LED
15     delay(1000);              // Espera 1 seg.
16 }
```

El sketch del listado 2.1 es similar al ejemplo **Blink** incluido en el IDE. Al ejemplo se le ha agregado la constante simbólica **LED** para definir el pin sobre el cual está conectado el LED, en este caso el pin dado por **LED_BUILTIN** (pin 13) correspondiente al LED incluido en la placa Arduino UNO.

Dentro del IDE Arduino

En el proceso de construcción del sketch que realiza la herramienta **arduino-builder**, cada archivo **.ino** de los sketches se convierten en archivos con extensión **.cpp**, que luego son compilados con el compilador de GCC en conjunto con los demás archivos fuentes (escritos en lenguaje C y C++) de las bibliotecas incluidas en el IDE. En uno de los archivos que se compila, de nombre **main.cpp**, se encuentra el cuerpo de la función **main()** e incluye además el archivo de cabecera **Arduino.h**.

El listado 2.2 muestra una versión simplificada del archivo **main.cpp**, donde se observan las llamadas a las funciones **setup()** y **loop()**.

Listado 2.2: Archivo `main.cpp` simplificado del IDE Arduino.

```
1 #include <Arduino.h>
2
3 int main(void)
4 {
5     // Llama funciones de inicialización del uC
6
7     setup();
8
9     for (;;) {
10         loop();
11         // Otras operaciones
12     }
13     return 0;
14 }
```

El listado 2.3 muestra un fragmento del archivo de cabecera `Arduino.h` de la biblioteca Arduino incluido en el archivo `main.cpp`. En este archivo es donde se definen las constantes simbólicas utilizadas para la programación del sketch, en conjunto con los prototipo de funciones de las funciones básicas como `pinMode()`, `digitalWrite()`, `delay()`, etc.

Listado 2.3: Archivo `Arduino.h` del IDE Arduino.

```
1 #ifndef Arduino_h
2 #define Arduino_h
3
4 #define HIGH 0x1
5 #define LOW 0x0
6
7 #define INPUT 0x0
8 #define OUTPUT 0x1
9
10 void pinMode(uint8_t, uint8_t);
11 void digitalWrite(uint8_t, uint8_t);
12 int digitalRead(uint8_t);
13 int analogRead(uint8_t);
14 void analogWrite(uint8_t, int);
15
16 unsigned long millis(void);
17 unsigned long micros(void);
18 void delay(unsigned long);
19 void delayMicroseconds(unsigned int us);
20
21 #endif
```

2.4.2. Programación sin el IDE Arduino

Es posible realizar la construcción y grabar los programas del microcontrolador ATmega328 de la placa Arduino UNO sin utilizar el IDE Arduino. Para ello es necesario instalar el “Toolchain” para la arquitectura particular del μ C, en este caso para arquitectura AVR. El toolchain está basado en el compilador GCC para AVR (`avr-gcc`) el cual en realidad es un compilador cruzado (cross-compiler). Un compilador cruzado es aquel que genera código máquina para una arquitectura diferente a aquella en la cual se ejecuta el compilador.

2. Programación del microcontrolador ATmega328

El Toolchain de GCC para AVR

El Toolchain de GCC para arquitectura AVR está compuesto por:

- **gcc**: el compilador de C y C++ [11]
- **binutils**: colección de herramientas para la manipulación de archivos binarios
- **libc-avr**: subconjunto de la biblioteca estándar de C con funciones específicas para arquitectura AVR [12]
- **avrdude**: programa para escribir y leer la memoria Flash del μ C

La instalación del Toolchain en Ubuntu se realiza instalando los siguientes paquetes: **gcc-avr**, **binutils-avr**, **avr-libc** y **avrdude**.

El programa del listado 2.4 hace encender y apagar un LED cada 1 segundo conectado a una salida digital, en este caso el bit 5 del Puerto B que, como puede verse en la figura 2.2, corresponde al LED incluido en la placa Arduino UNO. Antes del bucle infinito se configura este bit en particular como salida y en el bucle infinito se escribe un 1 (uno) y un 0 (cero) lógico cada 1 segundo.

Listado 2.4: Código fuente en lenguaje C, **blink.c**.

```
1 #include <avr/io.h>
2 #include <util/delay.h>
3
4 #define BLINK_DELAY_MS 1000
5
6 int main (void)
7 {
8     /* Configura el bit 5 del Puerto B como salida */
9     DDRB |= _BV(DDB5);
10
11    while(1)
12    {
13        /* Pone el bit 5 en alto (enciende el LED) */
14        PORTB |= _BV(PORTB5);
15        _delay_ms(BLINK_DELAY_MS);
16
17        /* Pone el bit 5 en bajo (apaga el LED) */
18        PORTB &= ~_BV(PORTB5);
19        _delay_ms(BLINK_DELAY_MS);
20    }
21    return 0;
22 }
```

Para construir el programa y grabarlo a la memoria del μ C ATmega328 de la placa Arduino UNO hay que realizar los siguientes pasos:

1. Construcción del código fuente (con extensión .c) hasta la etapa de ensamblado, de la cual se obtiene un archivo objeto (con extensión .o):

`avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p -c -o blink.o blink.c`

2. Enlazado con bibliotecas, del cual se obtiene un archivo binario (extensión .elf):

`avr-gcc -mmcu=atmega328p blink.o -o blink.elf`

3. Conversión del código binario (.elf) a formato objeto hexadecimal Intel (.hex)
`avr-objcopy -O ihex -R eeprom blink.elf blink.hex`
4. Grabación en la memoria de programa del μ C
`avrdude -F -V -c arduino -p ATMEGA328P -P /dev/ttyACM0 -b 115200 -U flash:w:blink.hex`

En (1) se realiza la construcción hasta la etapa de ensamblado (preprocesado, compilado y ensamblado) para convertir el archivo fuente con extensión .c al archivo objeto con extensión .o, donde: i) la bandera (flag) -c le indica que detenga la construcción antes de la etapa de enlazado, ii) la opción de optimización -Os realiza una optimización en el tamaño (size) del binario, iii) se define la constante simbólica F_CPU a 16000000UL para la función de retardo y iv) define el procesador con la opción -mmcu a atmega328p [13]. En (2) se enlaza el archivo objeto con extensión .o y se obtiene el archivo binario con extensión .elf (Executable and Linkable Format). En esta etapa también se enlaza el archivo objeto donde está el código de inicio (startup) del μ C, el cual realiza diferentes funciones de inicialización y llama a la función main() escrita en lenguaje C. En (3) se convierte el archivo binario del formato ELF con extensión .elf a formato HEX de Intel (opción -O) con extensión .hex. En (4) se graba el archivo HEX a la memoria Flash del μ C ATmega328 (opción -p) utilizando como programador la placa Arduino (opción -c) a través del puerto serie (opción -P) /dev/ttyACM0 a una velocidad de comunicación (opción -b) de 115200 bits por segundo.

Cabe aclarar que el código fuente del listado 2.4 puede ser transcrita dentro de un sketch utilizando el IDE Arduino. Para ello hay que colocar las líneas de código antes del bucle infinito dentro de la función setup() del sketch y lo que está dentro del bucle infinito en la función loop(). No es necesario incluir los archivos de cabecera, dado que ya son incluidos por la biblioteca del IDE Arduino. Se puede utilizar entonces el IDE Arduino para construir el programa y grabarlo a la memoria del μ C.

2.5. Programación de los puertos digitales

2.5.1. Programación de salidas digitales

El código fuente del listado 2.4 es un ejemplo de manejo de un puerto digital de entrada/salida. En este caso se utiliza el bit 5 del puerto B configurado como salida para encender y apagar un LED conectado al pin correspondiente. La placa Arduino UNO incluye un LED conectado a dicho pin (ver figura 2.2).

En la línea 9 del programa se configura el bit 5 del puerto B como salida modificando el valor del registro DDRB. Se utiliza el operador OR a nivel de bits para poner a 1 lógico dicho bit y así configurarlo como salida, como se describió en la sección 2.3.2. Dicha línea es equivalente a:

```
DDRB = DDRB | _BV(DDB5);
```

La macro _BV() (Bit Value) toma el valor del bit y lo convierte en un byte con dicho bit puesto a 1 y los demás a 0. Esta macro está definida como:

```
#define _BV(bit) (1 << (bit))
```

2. Programación del microcontrolador ATmega328

y DDB5 es una constante simbólica de valor 5. Por lo que la operación queda:

```
DDRB = DDRB | (1 << (5)); /* DDRB = DDRB | 0x20 */
```

con lo cual del registro DDRB tendrá el mismo valor que antes con el bit 5 puesto a 1.

Luego, en las líneas 14 y 19 se realiza una operación similar para poner a 1 y 0 lógico respectivamente el bit 5 del registro PORTB que, como se mencionó (sección 2.3.2), actúa directamente sobre el pin digital correspondiente, en este caso PB5. En la línea 14 se utiliza el operador OR lógico a nivel de bits para forzar a 1 un bit, mientras que en la línea 19 se utiliza el operador AND y NOT a nivel de bit para forzar a 0. En estas líneas se utiliza también la constante simbólica PORTB5 que vale 5. La operación que se realiza en la línea 19 es por lo tanto

```
PORTB = PORTB & ~(1 << (5)); /* PORTB = PORTB & 0xDF */
```

El programa del listado 2.5 muestra otra forma de hacer parpadear un LED utilizando el operador XOR (OR exclusivo).

Listado 2.5: Código fuente en lenguaje C, `blink_xor.c`.

```
1 #include <avr/io.h>
2 #include <util/delay.h>
3
4 #define BLINK_DELAY_MS 1000
5
6 int main (void)
7 {
8     /* Inicializa el pin digital como salida */
9     DDRB |= _BV(DDRB5);
10
11    while(1)
12    {
13        PORTB ^= _BV(PORTB5);      /* Toggle LED */
14        _delay_ms(BLINK_DELAY_MS);
15    }
16    return 0;
17 }
```

La figura 2.13 muestra un circuito utilizando la placa Arduino UNO a la que se le conectan 4 LEDs en los bits menos significativos del Puerto D, o sea en los pines PD0 a PD3. El listado 2.6 muestra el código fuente de un programa que muestra sobre los LEDs una cuenta binaria de 4 bits, o sea del rango de valores que va desde 0d = 0x0 = 0000b hasta 15d = 0xF = 1111b.

2.5.2. Programación de entradas digitales

La figura 2.14 muestra un circuito utilizando la placa Arduino UNO a la cual se le conecta un pulsador (switch) y un LED. El pulsador se conecta al pin 7 que es el bit 7 del puerto D (PD7) el cual debe configurarse como entrada y el LED se conecta al bit 1 del puerto B (PB1) que debe configurarse como salida (ver figura 2.2). El circuito utilizado es parte de la placa “ponchitoCIII” [14].

El programa del listado 2.7 maneja los puertos digitales del μ C para realizar la siguiente función: leer el valor de la entrada digital conectada al pulsador y en caso de estar presionado encender el LED durante 1 seg. y luego apagarlo.

2.5. Programación de los puertos digitales

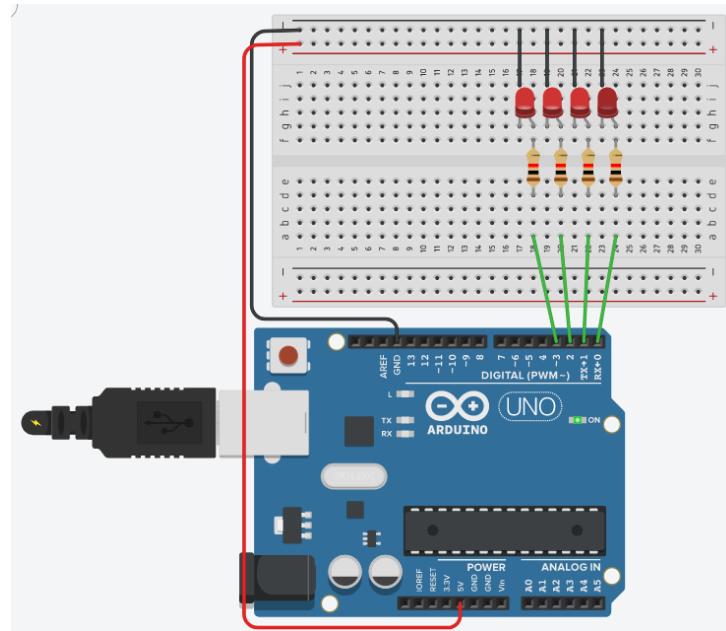


Figura 2.13: Circuito para el programa led_hex_couter.

Listado 2.6: Código fuente en lenguaje C, led_hex_couter.c.

```

1 #include <avr/io.h>
2 #include <util/delay.h>
3
4 #define LED_DDR    DDRD
5 #define LED_PORT   PORTD
6 #define LED_MASK   0x0F
7
8 #define DELAY_MS 500
9
10 int main (void)
11 {
12     unsigned char hex = 0;
13
14     /* Inicializa salidas digitales */
15     LED_DDR |= LED_MASK;
16
17     while(1)
18     {
19         LED_PORT &= ~(LED_MASK); // Apaga los LEDs
20         LED_PORT |= hex;        // Muestra valor
21
22         if(++hex > 16)      // Controla rango máximo
23             hex = 0;
24
25         _delay_ms(DELAY_MS);
26     }
27     return 0;
28 }
```

El programa comienza definiendo algunas constantes simbólicas de máscara y estado del pulsador. Luego, en la función `main()` se inicializan los puertos digitales de entrada/salida, configurando el bit del pulsador como entrada y el del LED como salida. Finalmente, en el bucle infinito se lee el valor de la entrada conectada al pulsador y en caso de estar

2. Programación del microcontrolador ATmega328

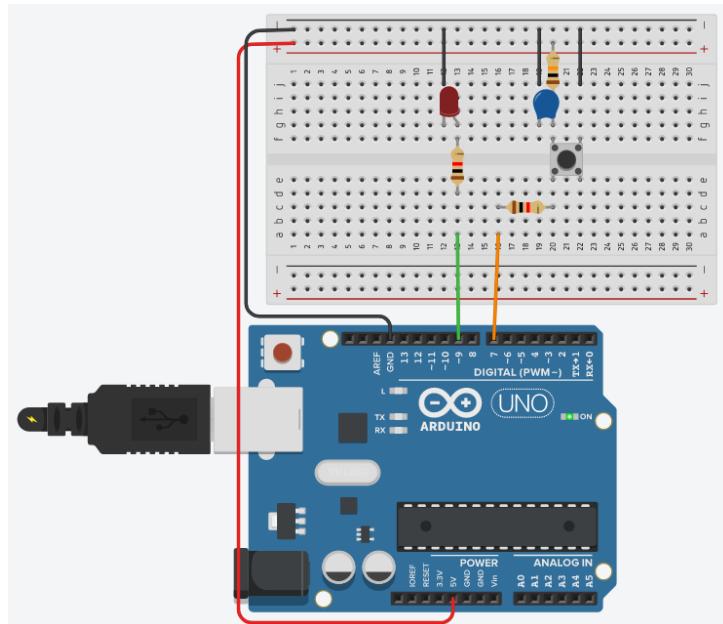


Figura 2.14: Circuito para el programa led_sw.

en estado de presionado se actúa sobre la salida conectada al LED.

La constante simbólica SW_MASK se utiliza como máscara para la lectura de la entrada digital (línea 23). La línea que realiza la lectura de la entrada es:

```
sw = (PIND & SW_MASK) >> PIND7;
```

donde se utiliza el valor del registro PIND para la lectura del Puerto D, la que se enmascara con SW_MASK, o sea que se ponen a cero todos los bits menos aquel cuyo valor interesa (a esto se le denomina máscara). Este valor luego se desplaza hacia la derecha para ocupar el bit menos significativo de la variable de 8 bits de nombre sw. Luego, se utiliza esta variable en la estructura de selección if para determinar si hay que encender el LED. La constante simbólica SW_PRES fija el valor que tendrá la entrada digital si el pulsador se encuentra presionado, lo cual dependerá de cómo este armado el circuito.

2.5.3. Constantes simbólicas y macros

Se muestra algunos detalles de la biblioteca de programación de los μ C AVR.

El archivo de cabecera `avr/io.h` incluye, mediante compilación condicional (usando la directiva de preprocesador `#if defined`), el archivo de cabecera particular del μ C utilizado, en este caso `avr/iom328p.h`. Este archivo contiene diferentes constantes simbólicas y macros para acceder a los registros del μ C. El listado 2.8 muestra una sección de este archivo de cabecera.

Como se observa, se definen los bits de los registros, tales como PINB1, DDR5, PORTB0, etc.; y los registros en sí mismo, tales como DDRB y PORTB.

La macro `_SFR_I08()` está definida en el archivo de cabecera `avr/sfr_defs.h` (Special Function Registers macros) como:

```
#define _SFR_I08(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
```

Listado 2.7: Código fuente en lenguaje C, led_sw.c.

```

1 #include <avr/io.h>
2 #include <util/delay.h>
3
4 /*
5  * LED: pin 9 --> PB1 (Salida)
6  * Sw: pin 7 --> PD7 (Entrada)
7 */
8 #define SW_MASK _BV(PIND7)
9 #define SW_PRES 0
10
11 int main(void)
12 {
13     uint8_t sw;
14
15     /* Configuración de entrada/salida digitales */
16     DDRB |= _BV(DDB1);      /* LED como salida */
17     DDRD &= ~_BV(DDD7);    /* Sw como entrada */
18     PORTB &= ~_BV(PORTB1); /* Apaga el LED */
19
20     while(1)
21     {
22         /* Lee el valor del pulsador */
23         sw = (PIND & SW_MASK) >> PIND7;
24
25         if(sw == SW_PRES)
26         {
27             PORTB |= _BV(PORTB1);    /* Enciende LED */
28             _delay_ms(1000);
29             PORTB &= ~_BV(PORTB1); /* Apaga LED */
30         }
31         _delay_ms(1);
32     }
33     return 0;
34 }
```

Listado 2.8: Sección del archivo de cabecera iom328p.h.

```

1 #define PINB _SFR_I08(0x03)
2 #define PINB0 0
3 #define PINB1 1
4 #define PINB2 2
5
6 #define DDRB _SFR_I08(0x04)
7 #define DDB0 0
8 #define DDB1 1
9 #define DDB2 2
10
11 #define PORTB _SFR_I08(0x05)
12 #define PORTB0 0
13 #define PORTB1 1
14 #define PORTB2 2
```

donde

```
#define _MMIO_BYTEmem_addr) (*(volatile uint8_t *)mem_addr))
```

y __SFR_OFFSET es una constante simbólica con la dirección de memoria base de los registros de funciones especiales. Por lo tanto, la macro _MMIO_Bytē() devuelve un puntero a la dirección de memoria de un registro en particular.

Por ejemplo, al hacer referencia al registro DDRB la macro se expande a

```
(*(volatile uint8_t *)((0x04) + 0x20))
```

o sea, un puntero que apunta a la dirección 0x24 (ver tabla 2.1). Acá puede observarse el uso del calificador `volatile` del lenguaje C.

2.6. Programación de la UART

Como se mencionó anteriormente, para poder utilizar la USART en una comunicación entre el μ C ATmega328 y otro dispositivo es necesario su inicialización, lo que consiste en configurar el formato de la trama y la velocidad de comunicación, además de habilitar el transmisor y/o receptor dependiendo de cada caso. En las secciones siguientes se muestra cómo configurar la USART0 del μ C ATmega328 para el modo asíncrono (UART), transmitir y recibir datos hacia y desde la PC. En el lado de la PC se puede utilizar ya sea el monitor serial del IDE Arduino o bien algún otro programa de terminal serial (ver sección 1.4.3).

2.6.1. Configuración de la USART

La configuración de la USART consiste en seleccionar el modo de funcionamiento asíncrono (UART), definir el tipo de trama y la velocidad de comunicación. El tipo de trama queda determinada por la cantidad de bits de datos, la cantidad de bits de parada y si se habilita o no la detección de errores mediante paridad y el tipo de paridad (par o impar) utilizada, como se describió en la sección 2.3.3. La velocidad de comunicación se refiere a la cantidad de bits por segundos (bps) o baud rates que son serializados en la transmisión/recepción de la trama. El formato de trama suele indicarse por una combinación de dos números y una letra donde:

- El primer número indica la cantidad de bits de datos.
- La letra indica el tipo de paridad, donde: N señala que no se utiliza el control de paridad, E (even) para la paridad par y O (odd) para impar.
- El último número indica la cantidad de bits de stop.

Un formato de trama muy utilizado es 8N1, o sea que la trama está conformada por 8-bits de datos, sin control de paridad y con 1-bit de parada. Otros ejemplos son: 5E2, 7E1, 601, etc.

Para la configuración de la USART0 del μ C ATmega328 se utilizan los registros de control y estado B y C, UCSRB y UCSRC (ver sección 2.3.3). Es importante también saber el estado por defecto que tienen los bits de estos registro al momento de realizar la configuración, como se observa en la figura 2.9.

El código fuente del listado 2.9 es un ejemplo de manejo de la USART0 del μ C ATmega328. En este programa se configura la USART0 para una trama 8N1 a una velocidad de 115.200bps. Según la tabla 2.2, para lograr esta velocidad de comunicación hay que escribir el valor 8 al registro UBRR0. El programa define una variable `ubrr` de 16-bits cuyo valor de inicialización está dado por la constante simbólica `BAUD_RATE_115200`. Luego, en

la sección de configuración (antes del bucle infinito), se configura la velocidad mediante las líneas:

```
UBRROH = (ubrr >> 8) & 0xFF;
UBRROL = ubrr & 0xFF;
```

donde en la primer línea se obtienen los 8-bits más significativos (MSB, Most Significant Byte) de la variable de 16-bits mediante una operación de desplazamiento hacia la derecha y una máscara, y el valor resultante se carga en el registro UBRROH (byte más significativo del registro UBRR); y en la segunda línea se obtiene los 8-bits menos significativos (LSB, Least Significant Byte) con una máscara y se carga en el registro UBRROL (byte menos significativo del registro UBRR). Las mismas operaciones puede utilizarse para otras velocidades de comunicación definidas por otras constantes simbólicas tales como:

```
#define BAUD_RATE_4800 207
#define BAUD_RATE_9600 103
#define BAUD_RATE_38400 25
#define BAUD_RATE_57600 16
```

Luego se configura la trama al formato 8N1 mediante el registro de control y estado C, UCSROC, con la siguiente línea:

```
UCSROC = _BV(UCSZ00) | _BV(UCSZ01);
```

lo que equivale a escribir el valor $6d = 00000110b = 0x06$. Aún cuando este es el valor por defecto del registro (ver figura 2.9), al configurarlo de forma explícita el programa queda mejor auto-documentado.

2.6.2. Transmisión de datos

Una vez configurada la velocidad de transmisión y el formato de la trama de comunicación, para que el μ C ATmega328 pueda enviar datos por la USART0 es necesario habilitar el transmisor. El transmisor se habilita escribiendo un 1 en el bit TXEN0 del registro UCSROB. Esto termina de configurar la USART0 para la transmisión, como se muestra en el programa del listado 2.9.

Este programa envía por puerto serie la cadena "Hola mundo" (incluido el carácter de nueva línea '\n' y retorno de carro '\r'). La transmisión de la cadena se realiza en el bucle principal del programa que se ejecuta cada 500ms utilizando un bucle **for**.

Para poder transmitir un dato por la USART hay que escribir el registro de transmisión de datos, UDRO (ver sección 2.3.3), habiendo verificado previamente que el búfer de transmisión está vacío. El estado del búfer de transmisión se verifica según el valor del bit UDRE0 del registro UCSROA. La línea:

```
while( !(UCSROA & _BV(UDRE0)) );
```

espera a que el búfer de transmisión este vacío. El estado de búfer se obtiene con la operación (UCSROA & _BV(UDRE0)) que devuelve 1 si el búfer está vacío. El bucle **while**, el cual no contiene sentencia (observar el punto y coma al final de la línea) se ejecuta mientras el búfer no este vacío. Cuando el búfer este vacío, la línea:

```
UDRO = dato[i];
```

escribe el siguiente carácter de la cadena a enviar en el registro de transmisión de datos, UDRO.

Listado 2.9: Código fuente en lenguaje C, `hola_mundo.c`.

```

1 #include <avr/io.h>
2 #include <util/delay.h>
3 #include <stdint.h>
4
5 #define BAUD_RATE_115200 8
6
7 int main()
8 {
9     int i;
10    uint16_t ubrr = BAUD_RATE_115200;
11    uint8_t dato[] = "Hola\u201c mundo\n";
12
13    /* Configuración el baud-rate */
14    UBRR0H = (ubrr >> 8) & 0xFF; /* Byte más significativo */
15    UBRR0L = ubrr & 0xFF;        /* Byte menos significativo */
16
17    UCSR0C = _BV(UCSZ00) | _BV(UCSZ01); /* UART con trama 8N1 */
18    UCSR0B |= _BV(TXEN0);                /* Habilita el transmisor */
19
20    while(1)
21    {
22        for(i = 0; dato[i] != '\0'; i++)
23        {
24            /* Espera a que el búfer de transmisión esté vacío */
25            while( !(UCSROA & _BV(UDRE0)) );
26            UDR0 = dato[i]; /* Escribe el búfer de transmisión */
27        }
28        _delay_ms(500);
29    }
30    return 0;
31 }
```

La figura 2.15 muestra la aplicación `cutecom` donde se observa la cadena recibida por la PC desde el μ C ATmega328.

2.6.3. Recepción de datos

La figura 2.16 muestra un circuito utilizando la placa Arduino UNO a la cual se le conecta un LED rojo y otro verde a las salidas digitales PB1 y PD5 respectivamente. El LED rojo cambiara de estado (encendido a apagado y apagado a encendido) cuando el μ C reciba el carácter 'r' o 'R'. Lo mismo para el LED verde al recibir el carácter 'v' o 'V'. El circuito utilizado es parte de la placa "ponchitoCIII" [14].

El programa del listado 2.10 maneja la USART del μ C para realizar la siguiente función: configura la USART0 en modo asíncrono (UART) con una velocidad de comunicación de 115.200bps y trama 8N1, habilita la recepción y actúa sobre los LEDs conectados a salidas digitales dependiendo del carácter recibido por la USART. La configuración de la USART0 se realiza como se describió en la sección 2.6.1 con la única diferencia de que en lugar de habilitar la transmisión de datos se habilita la recepción. Esto se realiza escribiendo un 1 en el bit RXEN0 del registro UCSROB. Además, se configuran como salida los pines digitales donde se encuentran conectados los LEDs.

En el bucle principal del programa se espera hasta recibir un carácter con la línea:

```
while( !(UCSROA & _BV(RXC0)) );
```

2.6. Programación de la UART

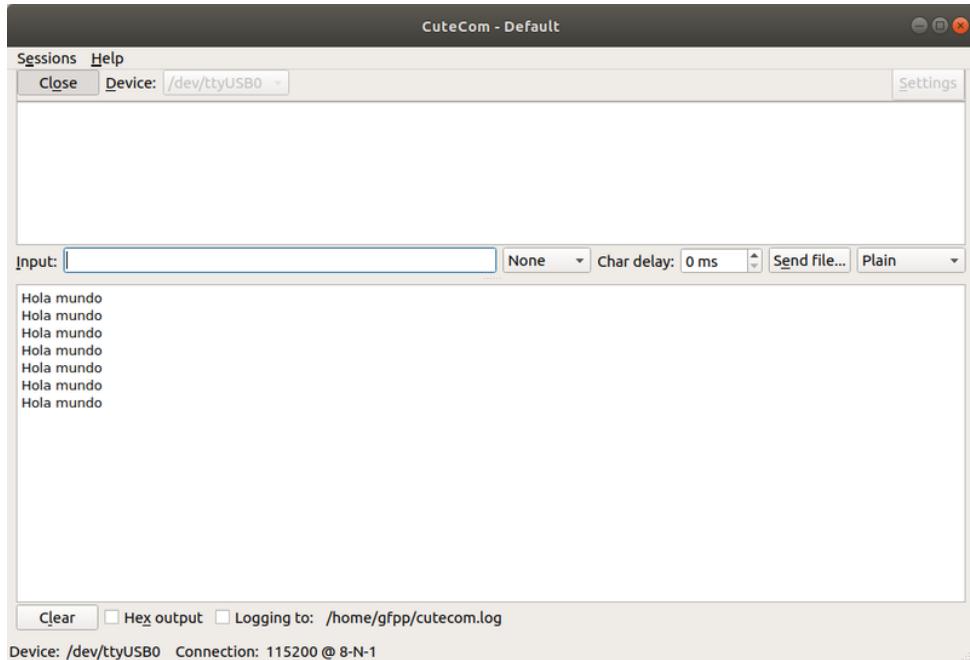


Figura 2.15: Aplicación `cutecom` con mensaje recibido.

de igual forma a cuando se espera que el búfer de transmisión este vacío para la transmisión de datos. Luego se utiliza la estructura de selección `switch-case` para actuar sobre el LED adecuado dependiendo del carácter recibido.

El programa del listado 2.11 configura la UART y habilita tanto la recepción como la transmisión de datos. Luego, espera un carácter y lo reenvía salvo que el carácter sea una

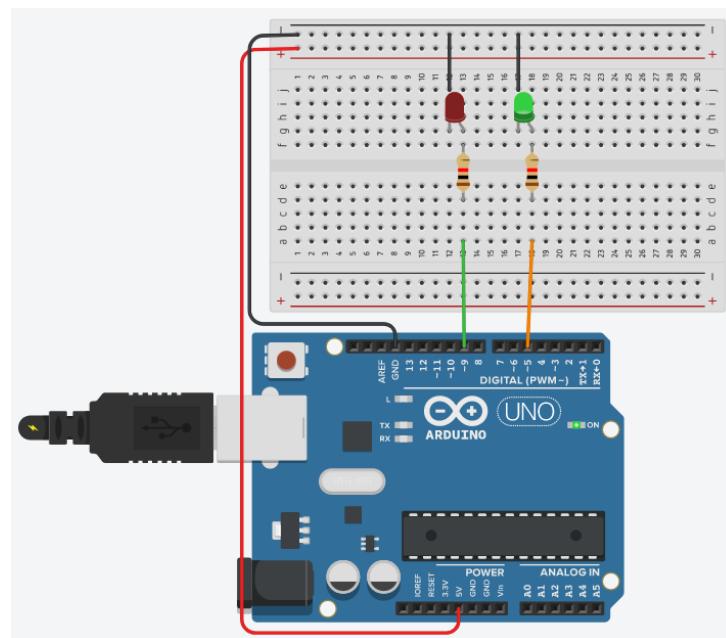


Figura 2.16: Circuito para el programa `uart_read`.

2. Programación del microcontrolador ATmega328

Listado 2.10: Código fuente en lenguaje C, `uart_read.c`.

```
1 #include <avr/io.h>
2 #include <stdint.h>
3
4 #define BAUD_RATE_115200 8
5
6 int main()
7 {
8     uint16_t ubrr = BAUD_RATE_115200;
9
10    DDRB |= _BV(DDB1);           /* Led rojo, salida */
11    DDRD |= _BV(DDD5);          /* Led verde, salida */
12
13    /* Configuración el baud-rate */
14    UBRR0H = (ubrr >> 8) & 0xFF; /* Byte más significativo */
15    UBRR0L = ubrr & 0xFF;        /* Byte menos significativo */
16
17    UCSROC = _BV(UCSZ00) | _BV(UCSZ01); /* UART con trama 8N1 */
18    UCSROB |= _BV(RXENO);          /* Habilita el receptor */
19
20    while(1)
21    {
22        /* Espera a recibir un dato */
23        while( !(UCSROA & _BV(RXC0)) );
24
25        switch(UDR0) {
26            case 'r':
27            case 'R':
28                PORTB ^= _BV(PORTB1); /* Cambia estado de LED rojo */
29                break;
30
31            case 'v':
32            case 'V':
33                PORTD ^= _BV(PORTD5); /* Cambia estado de LED verde */
34                break;
35        }
36    }
37    return 0;
38 }
```

letra en minúsculas (de 'a' a 'z'). En este caso, la letra se convierte a mayúsculas antes de reenviarla.

La figura 2.17 muestra la ventana de `cutecom` donde se observan las cadenas enviadas y recibidas hacia y desde el μ C ATmega328 convertidas a mayúsculas.

La conversión de caracteres de minúsculas a mayúsculas puede realizarse también utilizando funciones de la biblioteca estándar de C, como:

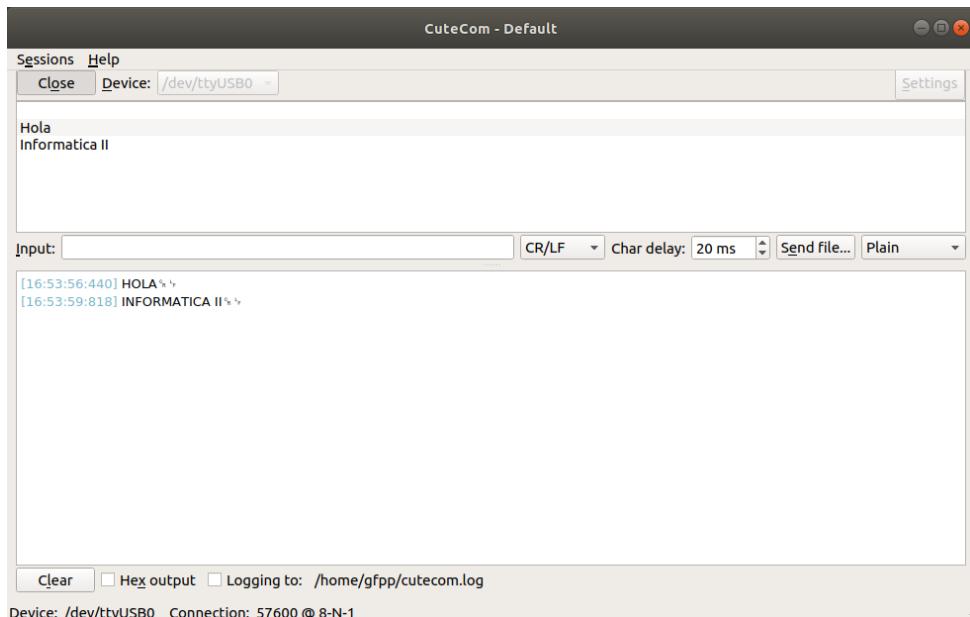
```
if( isalpha(car) )
    car = toupper(car);
```

para lo cual es necesario incluir el archivo de cabecera `ctype.h`.

Listado 2.11: Código fuente en lenguaje C, `uart_read_write.c`.

```

1 #include <avr/io.h>
2 #include <stdint.h>
3 #define BAUD_RATE_57600 16
4
5 int main()
6 {
7     uint16_t ubrr = BAUD_RATE_57600;
8     uint8_t car;
9
10    /* Configuración el baud-rate */
11    UBRR0H = (ubrr >> 8) & 0xFF; /* Byte más significativo */
12    UBRR0L = ubrr & 0xFF;
13    /* Byte menos significativo */
14    UCSR0C = _BV(UCSZ00) | _BV(UCSZ01); /* UART con trama 8N1 */
15    UCSR0B |= _BV(TXENO); /* Habilita el transmisor */
16    UCSR0B |= _BV(RXENO); /* Habilita el receptor */
17
18    while(1)
19    {
20        /* Espera a recibir un dato */
21        while( !(UCSROA & _BV(RXC0)) );
22        car = UDR0; /* Lee carácter */
23
24        if( (car >= 'a') && (car <= 'z') )
25            car -= 32;
26
27        /* Espera a que el búfer de transmisión esté vacío */
28        while( !(UCSROA & _BV(UDRE0)) );
29        UDR0 = car; /* Escribe el búfer de transmisión */
30    }
31    return 0;
32 }
```

Figura 2.17: Ejemplo del programa `uart_read_write`.

2. Programación del microcontrolador ATmega328

Capítulo 3

Comunicación entre PC y placa Arduino

A continuación se muestran algunos ejemplos de código fuente de programación de puerto serie para comunicar la PC con la placa Arduino UNO, como también algunos ejemplos de sketches Arduinos para tal fin. Luego se presentan ejercicios con desarrollos de programas del microcontrolador ATmega328, de la PC, o de ambos a la vez, algunos de los cuales incluyen la comunicación serie entre ellos.

En aquellos ejemplos y ejercicios que hace uso de la placa Arduino UNO utilizan circuitos basados en los presentados en las secciones 2.5 y 2.6 sobre programación del módulo GPIO y UART del microcontrolador ATmega328. Estos circuitos están basados a su vez en la placa “ponchitoCIII” [14].

3.1. Conexión entre la placa Arduino UNO y la PC

Al conectar una placa Arduino a la PC con sistema operativo GNU/Linux se genera un dispositivo serie, que generalmente se encuentra en `/dev/ttyACM0` o bien `/dev/ttyUSB0`. Este dispositivo serie se utiliza para grabar el programa al microcontrolador de la placa y puede usarse también para comunicar la PC con la placa Arduino.

Una manera de determinar el dispositivo donde se mapea el puerto serie al conectar una placa Arduino es a través del comando `dmesg`, el cual brinda información de Kernel del SO GNU/Linux.

Los listados 3.1 y 3.2 muestran la salida de comando `dmesg` al conectar la placa Arduino UNO Rev3 y Arduino UNO CH430, respectivamente. Como se observa, la placa Arduino UNO Rev3 genera el dispositivo `/dev/ttyACM0` mientras que la placa Arduino UNO CH430 el dispositivo `/dev/ttyUSB0`.

3. Comunicación entre PC y placa Arduino

Listado 3.1: Conexión de placa Arduino UNO Rev3.

```
> dmesg -H | tail
[Jul31 13:32] usb 1-1: new full-speed USB device number 29 using xhci_hcd
[+0,150543] usb 1-1: New USB device found, idVendor=2341, idProduct=0043
[+0,000005] usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=220
[+0,000004] usb 1-1: Manufacturer: Arduino (www.arduino.cc)
[+0,000003] usb 1-1: SerialNumber: 7563031343635141A1E2
[+0,004172] cdc_acm 1-1:1.0: ttyACM0: USB ACM device
```

Listado 3.2: Conexión de placa Arduino UNO CH430.

```
> dmesg -H | tail
[Jul31 14:08] usb 1-1: new full-speed USB device number 21 using xhci_hcd
[+0.148997] usb 1-1: New USB device found, idVendor=1a86, idProduct=7523
[+0.000002] usb 1-1: New USB device strings: Mfr=0, Product=2, SerialNumber=0
[+0.000002] usb 1-1: Product: USB2.0-Serial
[+0.002049] ch341 1-1:1.0: ch341-uart converter detected
[+0.000448] usb 1-1: ch341-uart converter now attached to ttyUSB0
```

3.2. Ejemplo de comunicación serie

3.2.1. Transmisión

A continuación se muestra el código fuente de un programa de PC para el envío de datos por puerto serie a una placa Arduino UNO, además del correspondiente sketch de Arduino para su puesta en funcionamiento.

El listado 3.3 muestra el código fuente de un sketch Arduino que permite encender y apagar LEDs según el carácter que se reciba por el puerto serie. El programa se puede probar utilizando el *Monitor Serie* del IDE Arduino para enviar los caracteres que hagan modificar el estado de los LEDs o bien se puede escribir un programa para tal fin.

Desde el punto de vista de la PC que envía los caracteres, al enviar la letra 'v' se conmuta el estado del LED verde conectado a PIN 5, la letra 'a' para el LED amarillo conectado al PIN 6 y la letra 'r' para el LED rojo conectado al PIN 9. En la sección 2.6.3 se muestra un programa similar sin hacer uso de las bibliotecas de programación de Arduino.

El listado 3.4 muestra el código fuente de un programa que envía el carácter 'v' por el puerto serie, que al ejecutar desde la PC conectada a la placa Arduino UNO y con el sketch anterior grabado, hace que el LED verde se encienda y apague cada un segundo aproximadamente. Se puede compilar el programa con

```
> gcc -Wall writechar.c termset.c -o writechar
```

el cual utiliza también el código fuente del módulo de configuración de puerto serie (ver la sección 1.5.3).

3.2.2. Recepción

El listado 3.5 muestra un código fuente de ejemplo de una aplicación de PC para leer el puerto serie mediante funciones de alto nivel de manejo de archivos. Este código se

Listado 3.3: Control de LEDs por puerto serie con Arduino (`ToggleLEDSerial.ino`).

```

1 /*
2  * Recibe una letra por puerto serie para cambiar el estado de los LEDS.
3  * 'v': cambia el estado del LED verde.
4  * 'a': cambia el estado del LED amarillo.
5  * 'r': cambia el estado del LED rojo.
6  *
7  * Se puede probar con el "Monitor serie" del IDE Arduino.
8  *
9  * El código de este ejemplo es de dominio público.
10 */
11
12 #define PIN_LED_VERDE      5
13 #define PIN_LED_AMARILLO   6
14 #define PIN_LED_ROJO        9
15
16 // La función 'setup' se ejecuta una única vez al presionar reset
17 // o encender la placa.
18 void setup() {
19     // Inicializa el puerto serie (UART) a 9600 bps.
20     Serial.begin(9600);
21
22     // inicialización de los pines GPIO como salida para los LEDs.
23     pinMode(PIN_LED_VERDE, OUTPUT);
24     pinMode(PIN_LED_AMARILLO, OUTPUT);
25     pinMode(PIN_LED_ROJO, OUTPUT);
26 }
27
28 // La función 'loop' corre indefinidamente una y otra vez.
29 void loop() {
30     size_t n;
31     char letra[1];
32
33     // Lee la letra (1 byte) por puerto serie.
34     n = Serial.readBytes(letra, 1);
35     if(n == 1)
36     {
37         switch(letra[0])
38         {
39             case 'v':
40                 toggle(PIN_LED_VERDE);
41                 break;
42             case 'a':
43                 toggle(PIN_LED_AMARILLO);
44                 break;
45             case 'r':
46                 toggle(PIN_LED_ROJO);
47                 break;
48         }
49     }
50 }
51
52 // Función para cambiar el estado de un LED (GPIO).
53 void toggle(uint8_t gpio)
54 {
55     if(digitalRead(gpio) == HIGH)
56         digitalWrite(gpio, LOW);
57     else
58         digitalWrite(gpio, HIGH);
59 }
```

puede utilizar para leer una cadena de texto enviada desde una placa Arduino al cual se le ha grabado el sketch de ejemplo que envía el valor del conversor analógico a digital

3. Comunicación entre PC y placa Arduino

Listado 3.4: Escritura de un carácter por puerto serie, `writetchar.c`.

```
1 #include <stdio.h>      /* Standard input/output definitions */
2 #include <fcntl.h>       /* File control definitions */
3 #include <unistd.h>      /* UNIX standard function definitions */
4 #include "termset.h"
5
6 int main(void)
7 {
8     int fd; /* Descriptor de archivo del puerto */
9     struct termios oldtty, newtty;
10
11    fd = open("/dev/ttyACM0", O_RDWR | O_NOCTTY | O_NDELAY);
12    if(fd == -1)
13    {
14        printf("ERROR: no se pudo abrir el dispositivo.\n");
15        return -1;
16    }
17    if(termset(fd, 9600, &oldtty, &newtty) == -1)
18    {
19        printf("ERROR: no se pudo configurar el TTY\n");
20        return -1;
21    }
22
23    tcflush(fd, TCIOFLUSH);
24    for(;;)
25    {
26        write(fd, "v", 1);
27        tcdrain(fd);
28        sleep(1);
29    }
30
31    close(fd);
32    return 0;
33 }
```

(ADC, Analog to Digital Converter). Este sketch se encuentra en el IDE Arduino en **File->Examples->01.Basics->AnalogReadSerial**. El esquema de la figura 3.1 muestra la conexión de un potenciómetro a la entrada analógica A0 (canal 0 del conversor analógico digital) para el programa mencionado.

El sketch **AnalogReadSerial** envía el valor del ADC como cadena (array de caracteres más fin de cadena). Esta cadena representa un valor entero en el rango de valores de 0 a 1023, dado que el ADC del μ C es de 10 bits ($2^{10} = 1024$). Un aspecto importante es que la cadena enviada es de longitud variable, dependiendo del número entero que representa,

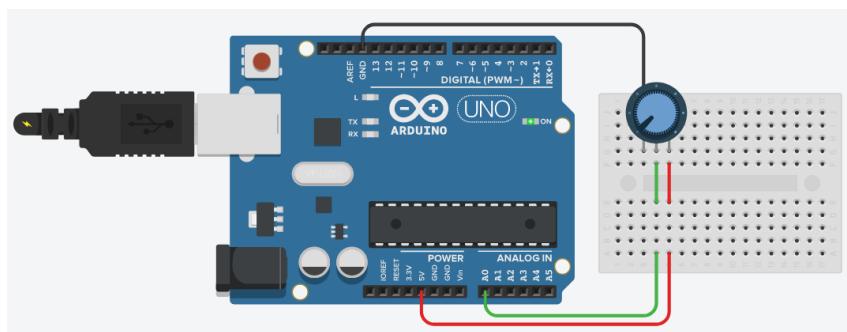


Figura 3.1: Conexión de potenciómetro a entrada analógica de la placa Arduino UNO.

por ejemplo: "0", "518", "1023", etc. Es por esto que el programa de comunicación serie de la PC que recibe los datos se realiza utilizando funciones de manejo de archivo de alto nivel. En este caso particular se utiliza la función `getline()`.

Listado 3.5: Lectura de archivo de alto nivel.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "termset.h"
4
5 int main(int argc, char * argv[])
6 {
7     int fd;
8     FILE * fptr;
9     ssize_t n_read;
10    size_t len = 0;
11    char * line;
12
13    if(argc != 3)
14    {
15        printf("USAGE: %s<device><baudrate>\n", argv[0]);
16        return -1;
17    }
18
19    fptr = fopen(argv[1], "r");
20    if(fptr == NULL)
21    {
22        printf("ERROR: opening %s file\n", argv[1]);
23        return -1;
24    }
25    fd = fileno(fptr);
26
27    if(termset(fd, atoi(argv[2]), &ttyold, &ttynew) == -1)
28    {
29        printf("ERROR: setting tty\n");
30        return -1;
31    }
32
33    for(;;)
34    {
35        n_read = getline(&line, &len, fptr);
36        if(n_read != -1)
37            printf("Line: %s", line);
38    }
39
40    fclose(fptr);
41    return 0;
42 }
```

El programa recibe mediante argumentos de la función `main()` el puerto serie y la velocidad de comunicación a utilizar. El mismo se puede compilar con

```
> gcc -Wall readline.c termset.c -o readline
```

generando el binario `readline`, el cual se debe ejecutar con

```
> ./readline /dev/ttyACM0 9600
```

3.3. Actividad práctica

Ejercicio 1.

Escribir un programa de PC para la comunicación serie con la placa Arduino con el sketch del listado 3.3 que:

- a) Abra el puerto serie /dev/ttyACM0 (o /dev/ttyUSB0 dependiendo de la placa) a 9600 bps (se puede utilizar el comando `stty` para ver la configuración por defecto del puerto serie y determinar si es necesario modificar la velocidad de comunicación).
- b) Muestre un menú en pantalla de forma continuada donde se solicite al usuario ingresar los siguientes caracteres: 'v' para cambiar el estado del LED verde, 'a' para el LED amarillo, 'r' para el LED rojo, hasta ingresar 's' para salir.
- c) Cierre el puerto serie.

Ejercicio 2.

El listado 3.6 muestra un sketch Arduino que envía por el puerto serie el estado de tres pulsadores que están conectados a los pines 4, 7 y 8. El estado de los pulsadores se envía como cadena con tres dígitos que pueden ser 1 o 0.

Escribir un programa de PC para la comunicación serie con la placa Arduino con sketch antes mencionado, el cual lea de forma continuada una cadena de 4 caracteres (3 estados de pulsadores y retorno de carro) y la imprime en la salida estándar.

Ejercicio 3.

El listado 3.7 muestra un sketch Arduino que toma el valor del ADC (Analog to Digital Converter) y lo envía por puerto serie en diferentes formatos. El μ C de la placa Arduino, al recibir el carácter 'c' envía el valor como cadena de caracteres de longitud fija, al recibir 'e' lo envía como número entero (4 bytes) y al recibir 'f' lo envía como número en punto flotante (4 bytes de la representación `float`).

Se puede utilizar el monitor serial del IDE Arduino para probar la recepción solo si se solicita el valor como cadena. Sin embargo, para observar los valores en representación como número entero o punto flotante es conveniente utilizar `cutecom` y ver los valores en representación hexadecimal, o bien escribir un programa para realizar la conversión de datos entre los tipos adecuados. El listado 3.8 muestra el código fuente para realizar la lectura por puerto serie de un valor entero (`int`) recibido en formato binario.

Escribir un programa de PC para la comunicación serie con la placa Arduino que:

- a) Abra el puerto serie pasado desde la línea de comandos a una velocidad de comunicación de 9600 bps.
- b) Muestre un menú de forma continuada donde se solicite al usuario ingresar los siguientes caracteres: 'c' para solicitar el valor del ADC como cadena, 'e' para solicitar el valor como entero, 'f' como punto flotante, e imprimir el valor recibido. O bien, 's' para terminar la ejecución del programa.

Listado 3.6: El estado de pulsadores por puerto serie (`ButtonStateSerialBin.ino`).

```

1 /*
2  * Envía los estados de las entradas (pulsadores) por puerto serie (UART).
3  * Se puede probar con el "Monitor serie" del IDE Arduino.
4  *
5  * El código de este ejemplo es de dominio público.
6  */
7
8 #define PIN_BOTON_IZQ 8
9 #define PIN_BOTON_MED 7
10#define PIN_BOTON_DER 4
11
12// La función 'setup' se ejecuta una única vez al presionar reset
13// o encender la placa.
14void setup() {
15    // Inicializa el puerto serie (UART) a 9600 bps.
16    Serial.begin(9600);
17
18    // Configura los GPIO de los pulsadores como entradas.
19    pinMode(PIN_BOTON_IZQ, INPUT);
20    pinMode(PIN_BOTON_MED, INPUT);
21    pinMode(PIN_BOTON_DER, INPUT);
22}
23
24// La función 'loop' corre indefinidamente una y otra vez.
25void loop() {
26    // Lee el valor de los pines de entrada (GPIO).
27    int boton_izq = digitalRead(PIN_BOTON_IZQ);
28    int boton_med = digitalRead(PIN_BOTON_MED);
29    int boton_der = digitalRead(PIN_BOTON_DER);
30
31    // Envía por puerto serie el estado del pulsador.
32    Serial.write(boton_izq + '0');
33    Serial.write(boton_med + '0');
34    Serial.write(boton_der + '0');
35    Serial.write('\n');
36    delay(100);
37}

```

c) Cierre el puerto serie.

Ejercicio 4.

La figura 3.2 muestra la conexión entre la placa Arduino UNO y un display de 7 segmentos. El display de 7 segmentos consiste de 7 diodos emisores de luz o LEDs (Light Emitted Diode), llamados segmentos, dispuestos para formar el número ocho. La mayoría de estos display tienen en realidad un total de 8 LEDs: 7 que forman el número y 1 para el punto decimal. Cada segmento se designa con la letra desde la A hasta la G, y DP para el punto decimal. Existe dos tipos de display de 7 segmentos: de ánodo común y de cátodo común.

Tanto en el esquema de conexión de la figura 3.2 como en la disposición de los pines de la figura 3.3 el display se corresponde a uno de cátodo común. La figura 3.3a muestra la posición de cada segmento y la figura 3.3b los pines de conexión de cada segmento.

El listado 3.9 muestra el código fuente de un sketch Arduino de un contador decimal (de 0 a 9) en el display de 7 segmentos del circuito de la figura 3.2.

Escribir un programa en lenguaje C con la misma funcionalidad del sketch antes mencionado, donde el módulo GPIO se programe mediante el acceso a los registros del μ C (ver sección 2.5.1).

3. Comunicación entre PC y placa Arduino

Listado 3.7: Lectura de ADC por puerto serie con Arduino (ADCFull.ino).

```
1 /*
2 * Envía por puerto serie el valor del ADC en diferentes formatos por petición.
3 * 'c': envía el valor entero como cadena ("0000" a "1023").
4 * 'e': envía el valor como nro. entero.
5 * 'f': envía el valor del voltaje como nro. de punto flotante (float).
6 *
7 * Se puede probar con el "Monitor serie" del IDE Arduino.
8 *
9 * El código de este ejemplo es de dominio público.
10 */
11
12 #define CANAL_ADC A0
13
14 // Unión de entero y unsigned char.
15 typedef union
16 {
17     uint8_t uc[4];
18     int i;
19 } int_uc_t;
20
21 // Union de float y unsigned char.
22 typedef union
23 {
24     uint8_t uc[4];
25     float f;
26 } float_uc_t;
27
28 // Variables globales.
29 size_t n;
30 uint8_t solicitud;
31
32 char cadena[5];    // Valor del ADC como cadena.
33 int_uc_t i_uc;    // Valor del ADC en entero.
34 float_uc_t f_uc; // Valor del ADC en punto flotante (float).
35
36 // La función 'setup' se ejecuta una única vez al presionar reset
37 // o encender la placa.
38 void setup() {
39     // Inicializa el puerto serie (UART) a 9600 bps.
40     Serial.begin(9600);
41 }
42
43 // La función 'loop' corre indefinidamente una y otra vez.
44 void loop() {
45     // Lectura de la entrada analógica.
46     int valor_adc = analogRead(CANAL_ADC);
47
48     // Lee la letra (1 byte) por puerto serie.
49     n = Serial.readBytes(&solicitud, 1);
50
51     if(n == 1)
52     {
53         switch(solicitud)
54         {
55             // Envía el valor del ADC como cadena.
56             case 'c':
57                 entero_a_cadena(valor_adc, 4, cadena);
58                 Serial.write(cadena, 5);
59                 break;
60             case 'e':
61                 i_uc.i = (int)valor_adc;
62                 Serial.write(i_uc.uc, 4);
63                 break;
64     }
```

Listado 3.7 (cont.): Lectura de ADC por puerto serie con Arduino (`ADCFull.ino`).

```
63     case 'f':
64         f_uc.f = valor_adc * (5.0 / 1023.0);
65         Serial.write(f_uc.uc, 4);
66         break;
67     }
68 }
69 delay(100);
70 }
71
72 void entero_a_cadena(unsigned int entero, unsigned char cant_digitos,
73   char * cadena)
74 {
75     unsigned int n, res;
76     for(n = 0; n < cant_digitos; n++)
77     {
78         res = (entero / potencia_entero(10, cant_digitos-n-1));
79         cadena[n] = res + '0';
80         entero -= res * potencia_entero(10, cant_digitos-n-1);
81     }
82     cadena[cant_digitos] = '\0';
83 }
84
85 int potencia_entero(int x, int n)
86 {
87     if(n == 0)
88         return 1;
89     int r = 1;
90     while(n--)
91         r *= x;
92     return r;
93 }
```

Ejercicio 5.

Agregar un pulsador al circuito de la figura 3.2 conectado a una entrada digital del μ C, por ejemplo PB0 (ver sección 2.5.2). Luego, modificar el programa del ejercicio anterior de forma tal que el valor mostrado en el display de 7 segmentos incremente su valor al presionar el pulsador en lugar de hacerlo de forma automática cada 1 seg. En el proceso de lectura de la entrada digital, el programa debe determinar si cambió de estado y si el nuevo estado se corresponde al pulsador presionado. Además analice:

- ¿Qué sucede si se mantiene presionado el pulsador?
- ¿Cómo debería modificarse el programa para que la cuenta se incremente si se mantiene presionado el pulsador?

Ejercicio 6.

Agregar un interruptor al circuito del ejercicio anterior conectado a una entrada digital (por ejemplo PB1) y modificar el programa para que el número mostrado en el display cambie al presionar el pulsador, incrementando o decrementando el valor según el estado del interruptor.

3. Comunicación entre PC y placa Arduino

Listado 3.8: Lectura de valor entero por puerto serie, `readint.c`.

```
1 #include <stdio.h>      /* Standard input/output definitions */
2 #include <fcntl.h>       /* File control definitions */
3 #include <unistd.h>      /* UNIX standard function definitions */
4 #include "termset.h"
5
6 typedef union
7 {
8     unsigned char uc[4];
9     int i;
10 } int_uchar_t;
11
12 int main(void)
13 {
14     int fd; /* Descriptor de archivo del puerto */
15     size_t n = 0;
16     unsigned char byte = 0;
17     struct termios oldtty, newtty;
18
19     int_uchar_t i_uc;
20
21     fd = open("/dev/ttyACM0", O_RDWR | O_NOCTTY | O_NDELAY);
22     if(fd == -1)
23     {
24         printf("ERROR: no se pudo abrir el dispositivo.\n");
25         return -1;
26     }
27     termset(fd, 9600, &oldtty, &newtty);
28
29     tcflush(fd, TCIOFLUSH);
30     for(;;)
31     {
32         write(fd, "e", 1);
33         tcdrain(fd);
34         usleep(250000);
35
36         n = read(fd, &(i_uc.uc[0]), 4);
37         printf("Valor entero: %d\n", i_uc.i);
38     }
39
40     close(fd);
41     return 0;
42 }
```

Ejercicio 7.

En el circuito de la figura 3.2, las salidas digitales conectadas a los segmentos A y B se corresponden con los pines 0 y 1 de la placa Arduino UNO, o sea, los bits PD0 y PD1 del Puerto D. Estos dos bits comparten la funcionalidad con los bits de transmisión (Tx) y recepción (Rx) del puerto serie del μ C ATmega328. Para poder utilizar el display de 7 segmentos en conjunto con el puerto serie se pide:

- a) Modificar el circuito de forma tal que los segmentos A y B pasen a estar conectados a las salidas digitales PB0 y PB1 respectivamente. O sea, cambiar PD0 por PB0 y PD1 por PB1.
- b) Modificar el programa del contador decimal para adecuarlo al nuevo circuito.

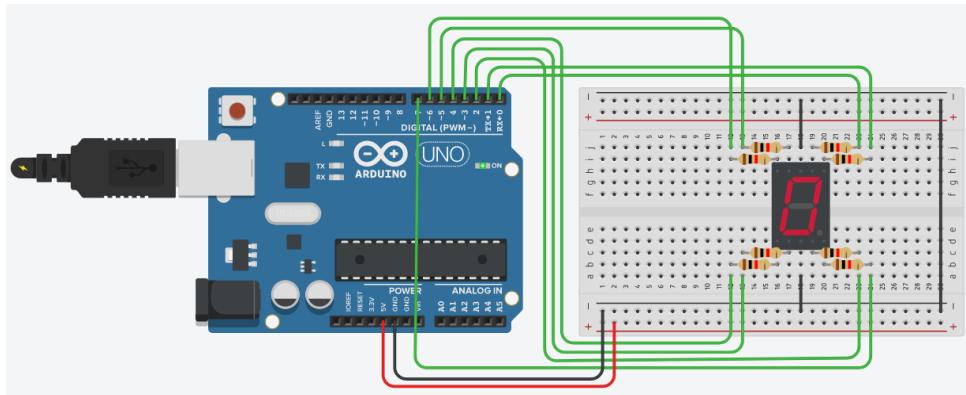
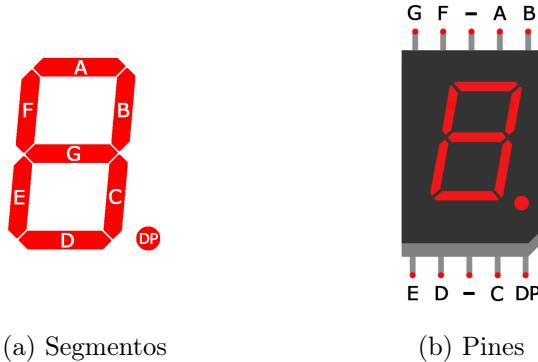


Figura 3.2: Circuito con display de 7 segmentos.



(a) Segmentos

(b) Pines

Figura 3.3: Display de 7 segmentos.

Ejercicio 8.

Escribir un programa en lenguaje C con la misma funcionalidad del sketch del listado 3.3, donde el módulo GPIO se programe mediante el acceso a los registros del μ C ATmega328.

Ejercicio 9.

Escribir un programa en lenguaje C con la misma funcionalidad del sketch del listado 3.6, donde el módulo GPIO se programe mediante el acceso a los registros del μ C ATmega328.

Ejercicio 10.

El listado 3.10 muestra el código fuente de un sketch Arduino que recibe por puerto serie un valor (byte) y lo visualiza en un display de 7 segmentos conectado a 8 pines digitales de salidas. La conexiones entre la placa Arduino UNO y el display de 7 segmentos son: segmento A con PB0, B con PB1, y los segmentos C hasta DP con PD2 hasta PD7.

Utilizar el programa `cutecom` para enviar datos de la PC a la placa Arduino. Para ello se debe configurar la entrada en modo hexadecimal (Seleccionar Hex en el menú desplegable ubicado al lado del campo de edición de entrada (`Input`)). Ver figura 1.10 de

3. Comunicación entre PC y placa Arduino

Listado 3.9: Contador decimal con display de 7 segmentos.

```
1 #define N_SEG 8
2 #define N_NUM 9
3
4 // Segments: A, B, C, D, E, F, G, DP
5 const unsigned char display_pin_num[] = {0, 1, 2, 3, 4, 5, 6, 7};
6
7 /*
8  0 | A, B, C, D, E, F      | 0011-1111 | 0x3F
9  1 | B, C                 | 0000-0110 | 0x06
10 2 | A, B, D, E, G       | 0101-1011 | 0x5B
11 3 | A, B, C, D, G       | 0100-1111 | 0x4F
12 4 | B, C, F, G          | 0110-0110 | 0x66
13 5 | A, C, D, F, G       | 0110-1101 | 0x6D
14 6 | A, C, D, E, F, G    | 0111-1101 | 0x7D
15 7 | A, B, C              | 0000-0111 | 0x07
16 8 | A, B, C, D, E, F, G | 0111-1111 | 0x7F
17 9 | A, B, C, F, G       | 0110-0111 | 0x67
18 */
19 unsigned char num_idx = 0;
20 const unsigned char display_num[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D,\n21                                     0x07, 0x7F, 0x67};
22
23 void setup()
24 {
25     unsigned char i;
26     for(i = 0; i < N_SEG; i++)
27         pinMode(display_pin_num[i], OUTPUT);
28     set_display(0x00);
29 }
30
31 void loop()
32 {
33     set_display(display_num[num_idx]);
34     if(++num_idx > N_NUM)
35         num_idx = 0;
36     delay(1000);
37 }
38
39 void set_display(unsigned char val)
40 {
41     unsigned char i;
42     for(i = 0; i < N_SEG; i++)
43     {
44         digitalWrite(display_pin_num[i], (val & 0x01));
45         val >>= 1;
46     }
47 }
```

la sección 1.4.3 (no puede utilizarse con el Monitor Serial del IDE Arduino dado que el mismo solo transmite datos en formato cadena).

- a) ¿Qué números en hexadecimal deben enviarse desde la PC al μ C para encender de a uno los diferentes segmentos del display?
- b) ¿Qué números en hexadecimal deben enviarse para representar de forma aproximada las diferentes letras del abecedario en el display?

Listado 3.10: Display de 7 segmentos y puerto serie.

```
1 #define N_SEG 8
2 unsigned char segment_pin_num[] = {8, 9, 2, 3, 4, 5, 6, 7};
3
4 void setup()
5 {
6     unsigned char i;
7     for(i = 0; i < N_SEG; i++)
8         pinMode(segment_pin_num[i], OUTPUT);
9     set_display(0x00);
10
11     Serial.begin(9600);
12 }
13
14 void loop()
15 {
16     unsigned char val;
17     if( Serial.available() > 0)
18     {
19         val = Serial.read();
20         set_display(val);
21     }
22     delay(100);
23 }
24
25 void set_display(unsigned char val)
26 {
27     unsigned char i;
28     for(i = 0; i < N_SEG; i++)
29     {
30         digitalWrite(segment_pin_num[i], (val & 0x01));
31         val >>= 1;
32     }
33 }
```

Ejercicio 11.

Escribir un programa en lenguaje C con la misma funcionalidad del sketch del listado 3.10, donde los GPIO y la UART se programen mediante el acceso a los registros del μC ATmega328.

Ejercicio 12.

Escribir un programa de PC que solicite al usuario ingresar un valor hexadecimal y luego lo envíe por el puerto serie, para reemplazar la terminal serial cutecom del ejercicio anterior. Este programa debe trabajar con el dispositivo serie que se le pasa como argumento en la línea de comandos. Además, debe leer el valor hexadecimal desde la entrada estándar utilizando la función `scanf` y terminar su ejecución si el valor ingresado es mayor a 0xFF.

Ejercicio 13.

Escribir un programa de PC que envíe una serie de valores por el puerto serie cada 1 seg. (el archivo de dispositivo se pasa por la línea de comandos). Los valores a enviar deben representar de forma aproximada las letras del abecedario en el display de 7 segmentos. Por ejemplo, utilizando la conexión de la placa Arduino UNO con el display de 7 segmentos

3. Comunicación entre PC y placa Arduino

como en los ejercicios anterior, se puede armar las siguientes letras: a con el valor 0x77, h con 0x76, n con 0x54, t con 0x78, etc.

Ejercicio 14.

Escribir un programa de PC que realice una petición de lectura del valor del ADC del µC ATmega328 grabado con el sketch del listado 3.7 en formato de punto flotante cada 100mseg. (utilizar la función usleep para generar el tiempo de espera), y muestre el valor leído por la salida estándar.

El programa debe recibir por línea de comando el dispositivo serie con el cual trabajar y la cantidad de dígitos decimales a utilizar para mostrar el valor. Por ejemplo, si el programa se ejecuta con

```
> ./read_adc /dev/ttyUSB0 2
```

se imprimen los valores con dos decimales como

2.21

2.21

2.33

2.45

y si se ejecuta con

```
> ./read_adc /dev/ttyUSB0 6
```

se imprimen los valores con seis decimales como

1.221896

1.041056

0.845552

0.703812

Luego, ejecutar el programa redireccionando la salida estándar a un archivo, tal como

```
> ./read_adc /dev/ttyUSB0 4 > datos.txt
```

Estos valores almacenados en un archivo puede utilizarse luego para generar una gráfica.

El listado 3.11 muestra un script de Python para graficar los datos del archivo `datos.txt`. Para usarlo hay que crear un archivo con extensión .py (p.e. `plot_txt.py`) en el mismo directorio donde se encuentra el archivo de datos, darle permisos de ejecución y ejecutar el script desde la línea de comandos.

Listado 3.11: Script de python para graficar archivo de datos.

```
1 #!/usr/bin/env python3
2
3 import numpy as np
4 from matplotlib import pyplot as plt
5
6 data = np.loadtxt('datos.txt')
7
8 plt.plot(data)
9 plt.show()
10 plt.grid()
```

Bibliografía

- [1] Dallas Semiconductor. Application Note 83. Fundamentals of RS-232 Serial Communications.
- [2] Atmega328 datasheet. <https://www.microchip.com/wwwproducts/en/ATMEGA328P>.
- [3] Arduino web page. <https://www.arduino.cc/>.
- [4] Hernando Barragán. The untold history of arduino. <https://arduinohistory.github.io/>.
- [5] Arduino uno rev3. <https://store.arduino.cc/usa/arduino-uno-rev3>.
- [6] Gonzalo Perez-Paina. Repositorio atmega328. <https://github.com/gfpp/atmega328>.
- [7] Autodesk. TinkerCAD. <https://www.tinkercad.com/>.
- [8] Processing web page. <https://processing.org/>.
- [9] Arduino IDE. <https://www.arduino.cc/en/Main/Software>.
- [10] Arduino libraries reference. <https://www.arduino.cc/reference/en/>.
- [11] Avrgcc. <https://nongnu.org/avr-libc/>.
- [12] Avr libc home page. <https://nongnu.org/avr-libc/>.
- [13] Gcc avr options. <https://gcc.gnu.org/onlinedocs/gcc/AVR-Options.html#AVR-Options>.
- [14] CIII, UTN-FRC. PonchitoCIII. <https://github.com/ciiiutnfrc/ponchitoCIII>.

Bibliografía
