



## TP IX – Entrada - Salida

### Envío y Recepción de archivos a través de dispositivo de Entrada - Salida

**Objetivo:** Desarrollar un programa que permita operar sobre dispositivos de entrada - salida del computador. Interactuar con el programa a través de la línea de comandos. Establecer distintos parámetros de comunicación.

#### **Introducción**

En un entorno UNIX los dispositivos están representados a través de archivos especiales ubicados en el sistema de archivos, al igual que cualquier archivo regular del usuario, esto permite un tratamiento uniforme, a través de una misma interfaz.

Los archivos especiales asociados a dispositivos se encuentran en `/dev`.

Los puertos seriales están representados por el archivo `ttyS0` (puerto serial 1), `ttyS1` (puerto serial 2) y así sucesivamente.

Los puertos usb están representados por el archivo `ttyUSB0` (puerto usb 1), `ttyUSB1` (puerto usb 2) y así sucesivamente.

Los puertos paralelos están representados por el archivo `lp0` (puerto paralelo 1), `lp1` (puerto paralelo 2), etc.; también suele existir el link<sup>1</sup> `/dev/printer`.

Por lo tanto, enviar un caracter a través del puerto serial 1 del computador implica grabar dicho caracter en el archivo `/dev/ttyS0`. Idem para la lectura, habrá que leer un caracter de dicho archivo.

La configuración de los parámetros de comunicación se realiza a través de una terminal (usando la función `tcsetattr()`), una vez que la terminal ha sido configurada, luego se asocia dicha terminal con el dispositivo a utilizar. Generalmente antes de cambiar la configuración de una terminal se guarda su configuración (usando la función `tcgetattr()`) actual para luego poder restablecerla<sup>2</sup>.

El Analista Y debe interactuar con varios dispositivos conectados a distintos puertos de los distintos computadores que tiene en cada una de las sucursales de la Empresa X. Por ejemplo, puede tratarse de puntos de venta que poseen impresoras fiscales conectadas al puerto paralelo del computador. En otra sucursal tiene un reloj de control de personal conectado a una workstation por

<sup>1</sup> Algo similar a los "shortcuts" de windows, es un fichero que apunta a otro fichero.

<sup>2</sup> En pruebas realizadas entre Linux y Windows, transfiriendo a través del puerto serie se pudo comprobar que al intentar restablecer la configuración original de la terminal (en el caso del programa de envío) ello provocaba la pérdida de datos transmitidos por parte del programa de recepción. Esto se pudo notar en archivos grandes y con equipos lentos. También las pruebas denotan una gran lentitud en la recepción/transmisión de datos a través del puerto serie en Windows, utilizando este mismo código recompilado con cygwin, posiblemente esto se deba a la intervención de `cygwin1.dll` que convierte las llamadas Unix en llamadas Windows. Debido a esto se incorporó el parámetro de delay para mejorar la sincronización de equipos con velocidad muy dispares que puedan afectar a la comunicación (a pesar de haberse configurado el puerto a la misma velocidad).



puerto serial. Otro sistema tiene una interfaz que interactúa a través de un puerto USB, etc.

La interacción con los dispositivos es muy variada y básicamente se trata del envío y recepción de caracteres a través de un puerto. En algunos casos, se trata de caracteres especiales que son interpretados de alguna forma por el receptor; en otros casos, son simples datos a enviar o recibir.

Otra variante son los parámetros de la comunicación: varían en cuanto a la cantidad de bits de datos, de paridad, en cuanto a la velocidad de comunicación con el dispositivo, en cuanto a los bits de stop y en cuanto al time-out de la comunicación.

Para solucionar estos problemas, el Analista Y ha pensado en el desarrollo de dos programas (uno para el envío y otro para la recepción) que interactúen con los puertos; los programas deberán:

- ◆ Recibir los siguientes parámetros por línea de comandos:
  - archivo (path) a enviar o recibir
  - nombre del dispositivo (/dev/...)
  - velocidad (número de baudios, ejemplo: 1200, 2400, ...)
  - time-out (número de segundos de time-out antes de dar por fallido el envío o recepción, ejemplo: 0, 1, 2, ...)
  - paridad (0 indica que no hay paridad, 1 indica paridad impar, 2 indica paridad par)
  - bits de datos por byte (5,6,7,8)
  - bits de stop por byte (1,2)
  - segundos de demora (segundos de demora luego de un envío o recepción, esto puede ser útil para sincronizar las distintas velocidades entre dispositivos, ejemplo: 0 (indica que no hay demora), 1 (demora de un 1 segundo),...)
- ◆ Configurar el puerto utilizando los parámetros indicados
- ◆ Realizar el envío o recepción del archivo indicado

La descripción anterior coincide con el siguiente diseño:



Programa de envío:

```
main()
  controlar parametros
  si parametros Ok entonces
    si pude abrir dispositivo en modo grabacion entonces
      si pude abrir archivo a enviar en modo lectura entonces
        guardar estado actual del dispositivo
        setear modo de control del dispositivo
        setear paridad
        setear bits de datos por byte
        setear modo de transmision
        setear time-out
        setear velocidad
        setear bits de stop por byte
        asignar al puerto los nuevos parametros
        bytes enviados = 0
        mientras haya podido leer un bloque de datos de archivo a enviar
          grabo en dispositivo los bytes leidos de archivo a enviar
          bytes enviados = bytes enviados + bytes leidos
          demoro <segundos de demora>
        fin mientras
        cierro dispositivo
        cierro archivo a enviar
        muestro la cantidad de bytes enviados
      sino
        error en apertura de archivo a enviar
      fin si
    sino
      error en apertura de dispositivo
    fin si
  sino
    error en parametros, mostrar forma de uso
  fin si
fin main()
```

Programa de recepción (muy similar al anterior):

```
main()
  controlar parametros
  si parametros Ok entonces
    si pude abrir dispositivo en modo lectura entonces
      si pude abrir archivo a recibir en modo grabacion entonces
        guardar estado actual del dispositivo
        setear modo de control del dispositivo
        setear paridad
        setear bits de datos por byte
        setear modo de transmision
        setear time-out
        setear velocidad
        setear bits de stop por byte
        asignar al puerto los nuevos parametros
        bytes recibidos = 0
        mientras haya podido leer un bloque de datos del dispositivo
          grabo en archivo a recibir los bytes leidos del dispositivo
          bytes recibidos = bytes recibidos + bytes grabados
          demoro <segundos de demora>
        fin mientras
        cierro dispositivo
```



```
        cierre archivo a recibir
        muestro la cantidad de bytes recibidos
    sino
        error en apertura de archivo a recibir
    fin si
sino
        error en apertura de dispositivo
    fin si
sino
        error en parametros, mostrar forma de uso
    fin si
fin main()
```

1. Implemente la librería `serial.c`. Evidentemente, ambos diseños tienen mucho en común, por lo tanto, sería buena idea "poner en un solo lugar todo el código a reusar", para ello, construiremos una librería formada por su archivo de código `serial.c` y su API `serial.h`. Todos los parámetros de comunicación se pueden especificar a través de la interfase de terminal de bajo nivel (ver capítulo 17 "Low-Level Terminal Interface" del manual de referencia de la librería GNU, `libc.pdf` incluido en el cd-rom de programación de esta asignatura), para ello se utiliza la estructura `struct termios`. Cambiar un parámetro implica modificar el contenido de algún elemento dentro de dicha estructura. Si el programa principal tendrá una referencia (puntero) a dicha estructura, la misma deberá ser pasada a cada una de las funciones de esta librería. Una API posible de esta librería podría ser:

```
void setParidad(struct termios *t,int par);
void setDataBits(struct termios *t,int bits);
void setTimeOut(struct termios *t,int sec);
void setVel(struct termios *t,char *velocidad);
void setStopBits(struct termios *t,int bits);
```

Dentro de la estructura `termios` existen una serie de atributos que estan formados por un conjunto de bits que representan determinadas características de la comunicación, el más importante de ellos es el atributo `c_cflag` que será utilizado para indicar el control de paridad, bits de datos, bits de stop y velocidad. ¿Cómo es posible indicar tantas cosas distintas en una única variable? A través de la activación y desactivación de determinados bits dentro del conjunto de bits que representa `c_cflag`. Por lo tanto, se requerirá del uso de operadores de manejo de bits. Tomemos como ejemplo el seteo de la paridad de la comunicación, los posibles valores del argumento par son:

Valor Argumento par	Significado
0	implica sin control de paridad
1	implica paridad impar
2	implica paridad par

Una forma de uso posible de esta función sería:

```
struct termios newtio;
int paridad;
...
// se asignan valores a newtio y paridad
...
setParidad(&newtio,paridad);
...
```



Para facilitar la activación/desactivación de bits, se puede utilizar los operadores:

Operador	Significado
	OR binario
&	AND binario
~	Negación/inversión binaria (donde hay un bit activado lo desactiva y viceversa)

Para facilitar la ubicación del conjunto de bits a cambiar, C los provee de constantes definidas en `termios.h`<sup>3</sup> que tienen algún significado lógico, a continuación se resumen algunas de las utilizadas en este proyecto<sup>4</sup>:

Constante	Significado	Aplicable sobre Atributo (de estructura <code>termios</code> ) / usado con función:
PARENB	Habilitar control de paridad	<code>c_cflag</code>
PARODD	Habilitar paridad impar	<code>c_cflag</code>
B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400, B57600, B115200, B230400 ... <sup>5</sup>	Velocidad en bps, 0 baudios (colgado), 50 baudios, 75, 110, ...	<code>c_cflag</code>
CSIZE	Mascara que representa todas las combinaciones posibles de bits de datos (5,6,7,8)	<code>c_cflag</code>
CS5, CS6, CS7, CS8	Habilitar 5,6,7 u 8 bits de datos	<code>c_cflag</code>
CSTOPB	Habilitar 2 bits de stop (caso contrario, se asume 1 bit de stop)	<code>c_cflag</code>
CLOCAL	Indica que la terminal esta conectada localmente (ignora la linea de estado del modem)	<code>c_cflag</code>
CREAD	Habilita el hecho de que la entrada pueda ser leída desde la terminal, caso contrario, la entrada será descartada ni bien llegue.	<code>c_cflag</code>
ICANON	Habilita modo canónico de comunicación <sup>6</sup> (caso contrario se	<code>c_lflag</code>

<sup>3</sup> En plataformas Windows utilizando cygwin se encuentran definidas dentro del archivo `\usr\include\termios.h`. En linux `/usr/include/termios.h` lo redirecciona a `/usr/include/bits/termios.h`.

<sup>4</sup> Para mayor información, favor de consultar el Capítulo 17 "Low-Level Terminal Interface" del "The GNU C Library Reference Manual".

<sup>5</sup> Las velocidades varían según la plataforma, por ejemplo, en Windows utilizando cygwin se cuenta con B128000, B256000; mientras que en Linux éstas no existen. En Linux se cuenta con B460800, B500000, B576000, B921600, B1000000; mientras que en Windows éstas no existen.

<sup>6</sup> Los sistemas POSIX soportan dos modos de entrada: canónico o no canónico. El modo de procesamiento canónico



	asumirá modo no canónico)	
ECHO	Habilita el echo de los caracteres recibidos en la terminal	c_lflag
INPCK	Habilita el chequeo de paridad (caso contrario, no habrá control de paridad)	c_iflag (para input) c_oflag (para output)
PARMRK	Los bytes con error de paridad serán marcados (se usa en combinación con INPCK) agregando previamente dos bytes adicionales (0377 y 0)	c_iflag (para input) c_oflag (para output)
ISTRIP	Los bytes recibidos correctamente serán representados con 7 bits (caso contrario, se usarán 8 bits).	c_iflag (para input) c_oflag (para output)
TCIFLUSH	Limpia la cola de input asociada con la terminal.	tcflush()
TCOFLUSH	Limpia la cola de output asociada con la terminal.	tcflush()
TCIOFLUSH	Limpia la cola de input y output asociada con la terminal.	tcflush()
TCSANOW	Setea los atributos de la terminal ahora, inmediatamente (otras constantes pueden indicar que esto se haga en forma diferida).	tcsetattr()

No debemos olvidar que atributos tales como `c_iflag`, `c_oflag`, `c_cflag`, `c_lflag`, etc. tienen un conjunto de bits activados/desactivados y que, para cambiar algún aspecto de la comunicación *debemos ser cuidadosos de sólo afectar al conjunto de bits adecuado*. Tomemos como ejemplo a la función `setParidad()`, esta función primero deberá determinar si hay control de paridad (ver tabla argumento `par`):

```
void setParidad(struct termios *t,int paridad) {
    // seteo paridad
    if (!paridad) { // deshabilito control de paridad
        t->c_cflag &= ~PARENB;
    } else { // habilito control de paridad
        t->c_cflag |= PARENB;
    }
    ...
}
```

El uso de un operador de la forma: `<variable> <operador> = <expresion>`; es equivalente a `<variable> = <variable> <operador> <expresion>`. Por lo tanto, `a+=b` equivale a: `a=a+b`. La expresión `t->c_cflag &= ~PARENB` equivale a `t->c_cflag = t->c_cflag & (~PARENB)`.

implica que la terminal interprete determinados caracteres de control y actúe en consecuencia (por ejemplo, no se leerá el input de la terminal hasta que el usuario presione la tecla de Intro). En modo no canónico, los caracteres no se agrupan en líneas, son un simple flujo de bytes, sobre el cual no se realiza ningún tipo de interpretación.



¿Por qué debo hacer: `t->c_cflag &= ~PARENB;` para deshabilitar el control de paridad? ¿Por qué no hacer otra instrucción? Investiguemos. Observemos el contenido de `termios.h`:

```
...
#define NCCS 18
...
#define PARENB 0x00100
...
typedef unsigned char cc_t;
typedef unsigned int tcflag_t;
typedef unsigned int speed_t;
typedef unsigned char ospeed_t;
...
struct termios
{
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    char c_line;
    cc_t c_cc[NCCS];
    ospeed_t c_ispeed;
    ospeed_t c_ospeed;
}
...
```

El valor definido para `PARENB` es `0x00100` (hexadecimal), en decimal equivale a 256, en binario `100000000` (el noveno bit activado, el resto desactivado); es decir, que activar el control de paridad equivale a activar el noveno bit dentro de `c_cflag`. El valor `0x00100` es de tipo entero (`int`) al igual que `tcflag_t`, por lo tanto, `PARENB` y `c_cflag` son tipos compatibles.

La operación a realizar debería ser tal, que solo se afectara al noveno bit, dejando todo lo demás igual que antes dentro `c_cflag`. Supongamos que `c_cflag` vale 4 (100 en binario), hagamos las cuentas:

```
PARENB      100000000
~PARENB     011111111
c_cflag     000000100

      c_cflag   000000100
&    ~PARENB  011111111
      -----
              000000100
```

Podemos observar que el cálculo permite desactivar el noveno bit de `c_cflag` sin alterar el resto de su contenido (100), pero claro, antes de comenzar, el noveno bit de `c_cflag` ya estaba desactivado. Probemos ahora con el noveno bit activado en `c_cflag`:



```
PARENB      100000000
~PARENB     011111111
c_cflag     100000100

      c_cflag     100000100
&      ~PARENB     011111111
      -----
              000000100
```

El resultado es el mismo, esto prueba que estando o no activado previamente el control de paridad, la sentencia `t->c_cflag &= ~PARENB;` lo desactiva, sin alterar el resto de los bits de `c_cflag`.

El mismo análisis puede hacerse para la activación del bit de control de paridad: `t->c_cflag |= PARENB;` . Comencemos con `c_cflag` valiendo 4 (100):

```
PARENB      100000000
c_cflag     000000100

      c_cflag     000000100
|      PARENB     100000000
      -----
              100000100
```

Podemos comprobar que activa el noveno bit sin alterar el resto. Ahora probemos con el noveno bit activado en `c_cflag`:

```
PARENB      100000000
c_cflag     100000100

      c_cflag     100000100
|      PARENB     100000000
      -----
              100000100
```

El noveno bit de `c_cflag` continua activado y el resto de los bits no se han cambiado. Ahora Ud. ya puede comprender las operaciones binarias básicas que requiere este programa, veamos el código completo de `setParidad()`:

```
void setParidad(struct termios *t,int paridad) {
    // seteo paridad
    if (!paridad) {
        //PARENB significa habilitar bit de paridad
        //entonces, esto deshabilita el bit de paridad:
        t->c_cflag &= ~PARENB;
    } else { // habilito paridad
        t->c_cflag |= PARENB;
    }
}
```





```
    }  
    if ( paridad == 1 ) { // paridad impar  
        //PARODD habilita paridad impar  
        t->c_cflag |= PARODD;  
    } else { // paridad par, implica no paridad impar  
        t->c_cflag &= ~PARODD;  
    }  
}
```

Otras funciones más que son parte de la biblioteca serial.c:

```
...  
// <stop bits>=1,2 (nro. de bits de stop)  
void setStopBits(struct termios *t,int stopbits) {  
    //CSTOPB indica 2 bits de stop, caso contrario es solo 1 bit de stop  
    if ( stopbits == 2 ) t->c_cflag |= CSTOPB;  
    else  
        t->c_cflag &= ~CSTOPB;  
}  
  
// bits=5,6,7,8 (nro. de bits por byte)  
void setDataBits(struct termios *t,int bits) {  
    //CSIZE es una mascara para todos los tamaños de datos en bits,  
    //entonces, haciendo un and con su negacion, elimina el  
    //seteo actual en cuanto al tamaño de datos en bits  
    t->c_cflag &= ~CSIZE;  
    switch(bits) {  
        case 5:  
            t->c_cflag |= CS5;  
            break;  
        case 6:  
            t->c_cflag |= CS6;  
            break;  
        case 7:  
            t->c_cflag |= CS7;  
            break;  
        case 8:  
            t->c_cflag |= CS8;  
            break;  
        default:  
            t->c_cflag |= CS8;  
    }  
}  
  
// nro.de segundos de time out en input/output (0 indica no time out)  
void setTimeout(struct termios *t,int sec) {  
    t->c_cc[VTIME] = sec*10; /* inter-character timer unused */  
}  
...  
}
```

2. Continuamos ampliando la librería serial.c. Falta implementar setVel() para establecer la velocidad en la comunicación. Obsérvese que el argumento de la velocidad es una cadena de caracteres pero las constantes definidas en termios.h son de tipo int. El usuario ingresará en la línea de comandos "2400" y ello internamente deberá transformarse en B2400 (valor int definido dentro de termios.h). Para resolver esto, podemos usar un "traductor" que pase de una cadena a su valor entero correspondiente, para ello podemos definir dentro de serial.h un nuevo tipo de dato que ayudará a la conversión:



```
...
typedef struct velocidad {
    char *nombre;
    int valor;
} velocidad;
...
```

Dentro de `serial.c` podemos crear un arreglo global o estático a esta librería de tipo `velocidad` en donde se indiquen las velocidades soportadas por esta librería:

```
...
// dato global de biblioteca
velocidad vel[] = {
    {"0",B0}, {"50",B50}, {"75",B75}, {"110",B110}, {"134",B134}, {"150",B150},
    {"200",B200}, {"300",B300}, {"600",B600}, {"1200",B1200}, {"1800",B1800},
    {"2400",B2400}, {"4800",B4800}, {"9600",B9600}, {"19200",B19200},
    {"38400",B38400}, {"57600",B57600}, {"115200",B115200}, {"230400",B230400},
    { '\0', 0 }
};
...
```

Cada pareja de valores dentro del arreglo de tipo `velocidad` se corresponden con los valores de nombre y valor. Por ejemplo, `{"0", "B0"}` indica que `vel[0].nombre` apunta a "0" y `vel[0].valor` vale B0 (que en `terminos.h` esta definido como un entero que vale `0x00000`) y así sucesivamente. La última pareja de valores `{ '\0', 0 }` asignará un valor nulo (NULL, '\0', valor decimal 0) en nombre para indicar el final de las velocidades posibles. Usando este arreglo se puede implementar una función que haga una búsqueda secuencial dentro del mismo a partir de "2400" y retorne la posición (subíndice) en que se encuentra dentro del arreglo `vel[]` (se implementa usando punteros):

```
...
// busca strVel en tabla de velocidades,
// si no esta devuelve -1
// si esta devuelve su posicion
int findVel(char *strVel) {
    velocidad *v = &vel[0];
    while(v->nombre) {
        if (strcmp(v->nombre, strVel) == 0) return v - &vel[0];
        v++;
    }
    return -1;
}
...
```

La función `findVel()` será llamada desde la función `setVel()`.

Para indicar la velocidad de comunicación se puede usar también el flag `c_cflag` y/o las funciones `cfsetispeed()`, `cfsetospeed()` (para las velocidad de input y output respectivamente):



```
...  
// <velocidad>=<baudios>  
void setVel(struct termios *t,char *velocidad) {  
    int vpos = findVel(velocidad);  
    int ret;  
    if ( vpos != -1) {  
        t->c_cflag |= vel[vpos].valor;  
        ret=cfsetispeed(t, vel[vpos].valor);  
        ret=cfsetospeed(t, vel[vpos].valor);  
    } else printf("%s baudios no es una velocidad correcta!\n",velocidad);  
}  
...
```

3. Ya tendríamos una biblioteca básica para comunicaciones. Ahora vamos a comenzar por el programa de envío (sendserial.c), cuyo diseño se indicó al comienzo. El usuario va a interactuar con el programa a través de la línea de comandos, es decir, que el programa podría ejecutarse de la siguiente forma en Linux utilizando un shell script:

```
#!/bin/bash  
#envio desde linux debian 5.01  
./sendserial serial.c /dev/ttyS0 9600 5 1 7 1 2
```

Algo similar sería en Windows, utilizando un archivo .bat<sup>7</sup>:

```
@echo off  
rem Envio desde windows  
sendserial.exe serial.c /dev/ttyS0 9600 10 1 7 1 0
```

Los argumentos de la línea de comandos se representan y acceden en C a través de los argumentos declarados en la función main(). Por ejemplo, si se declara la función main() como:

```
...  
int main(int argc,char *argv[]) {  
    ...  
}  
...
```

Para los ejemplos de ejecución anterior, el valor de argc será de 9 y el contenido apuntado por argv[] serían las siguientes cadenas de caracteres (para el caso de Linux):

<sup>7</sup> Para un programa compilado con cygwin y que éste tenga acceso a cygwin1.dll.



Posición argv[]	Contenido argv[]	Significado
argv[0]	sendserial	Nombre del Programa
argv[1]	serial.c	Archivo a enviar/recibir
argv[2]	/dev/ttyS0	Dispositivo a utilizar en envío/recepción
argv[3]	9600	Velocidad
argv[4]	5	segundos de time-out en envío/recepción (0..n)
argv[5]	1	Paridad (0,1,2)
argv[6]	7	Bits de datos (5,6,7,8)
argv[7]	1	Bits de stop (1,2)
argv[8]	2	Segundos de demora entre cada envío/recepción (0..n)

En cuanto al diseño del programa "controlar parámetros" se refiere a controlar los valores posibles de `argc` y cada uno de los elementos de `argv[]`.

Hasta incluso, si el usuario no indica ningún argumento (`argc == 1`) o el número de argumentos fuese distinto de 9, se podría mostrar una ayuda en cuanto a los argumentos soportados.

"Abrir dispositivo en modo grabación", implica declarar e instanciar un descriptor de archivo (file descripto, valor entero que indica el número de archivo abierto por este proceso):

```
...
int fd = open(argv[2], O_WRONLY | O_NOCTTY );
if (fd < 0) {
    fprintf(stderr, "Error abriendo dispositivo [%s]\n", argv[2]);
    exit(-1);
} else { // pude abrir dispositivo Ok
    ...
}
...
```

Para abrirlo en modo lectura, indicar flags: `O_RDONLY | O_NOCTTY`

Para "abrir archivo a enviar" (en este caso, apertura de archivo como stream buffered a ser enviado):

```
...
FILE *fpi = fopen(argv[1], "rb");
if ( fpi == NULL ) {
    perror(argv[1]);
    exit(-2);
}
...
```

Para abrirlo en modo escritura, el segundo argumento de `fopen()` podría ser "wb" (write binary).

Para "guardar el estado actual del dispositivo":

```
...
struct termios oldtio;
...
// guardar estado actual del puerto (fd previamente abierto)
tcgetattr(fd, &oldtio);
...
```

Para "setear el modo de control del dispositivo":



```
...
    struct termios oldtio;
...
    // guardar estado actual del puerto (fd previamente abierto)
    tcgetattr(fd, &oldtio);
...
```

Podemos usar a `newtio` para cargar allí todos los nuevos seteos y luego configurar el puerto con esta nueva configuración:

```
...
    struct termios newtio;
...
    // guardar estado actual del puerto (fd previamente abierto)
    tcgetattr(fd, &newtio);
...
```

Para "setear el modo de control del dispositivo":

```
...
    newtio.c_cflag = CLOCAL | CREAD;
...
```

Para "setear paridad":

```
...
    // seteo paridad
    setParidad(&newtio, atoi(argv[5]));
...
```

Para "setear bits de datos por byte":

```
...
    // bits por byte
    setDataBits(&newtio, atoi(argv[6]));
...
```

Para "setear modo de transmisión" (no canónico, no echo) en el caso de envío:

```
...
    // set input mode (non-canonical, no echo, ...)
    newtio.c_oflag = 0;
    newtio.c_lflag &= ~(ICANON|ECHO);
...
```

Idem en el caso de recepción:

```
...
    /* set input mode (non-canonical, no echo, ...) */
    newtio.c_iflag = INPCK | PARMRK | ISTRIP;
    newtio.c_lflag &= ~(ICANON|ECHO);
...
```



Para "setear time-out":

```
...
// set time out
setTimeout(&newtio, atoi(argv[4]));
...
```

Para "setear velocidad":

```
...
// seteo velocidad
setVel(&newtio, argv[3]);
...
```

Para "setear bits de stop por byte":

```
...
// seteo bits de stop
setStopBits(&newtio, atoi(argv[7]));
...
```

Para "asignar al puerto los nuevos parámetros":

```
...
// paso los nuevos parametros al file descriptor del puerto
tcflush(fd, TCOFLUSH);
tcsetattr(fd, TCSANOW, &newtio);
...
```

El loop principal del programa de envio:

```
...
char buf[255];
size_t nread=0;
...
// demora
int sdelay = atoi(argv[8]);
if ( sdelay < 0 ) sdelay =0;

// leo archivo y envio a puerto serie
while( (nread=fread(buf,1,255,fpi)) ) {
    nwrite+=write(fd,buf,nread);
    sleep(sdelay);
}
tcflush(fd, TCOFLUSH);
...
```

Para "cerrar dispositivo y archivo":

```
...
// cierro archivo y puerto
close(fd);
fclose(fpi);
...
```



Los archivos de cabecera a utilizar serían:

```
...  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <termios.h>  
#include "serial.h"  
...
```

Compile el programa como: `gcc -Wall -o senserial sendserial.c serial.c`

4. Idem anterior, pero ahora desarrolle el programa de recepción (`receiveserial.c`).

Compile el programa como: `gcc -Wall -o receiveserial receiveserial.c serial.c`

5. Si cuenta con un cable serial (par cruzado) como el utilizado para las terminales seriales Unix, puede utilizar estos programas para enviar y recibir archivos a través del puerto serial<sup>8</sup>. Si recompila los programas en Windows utilizando cygwin, podría hacerlo entre equipos con distintos Sistemas Operativos.

### **Bibliografía**

- ◆ Stallings, William, "Sistemas Operativos", 5ta. Edición. Prentice Hall. 2001. Madrid.
- ◆ Stevens, Richard, "Advanced Programming in the UNIX Environment", Addison-Wesley Professional Computing Series, 1993, ISBN 0-201-56317-7
- ◆ LooseSandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, Ulrich Drepper, "The GNU C Library Reference Manual", Free Software Foundation, 2007, Boston, USA.

*Espero que esta práctica haya sido de vuestro agrado y le permita comprender mejor los mecanismos básicos de manejo de dispositivos y su forma de configuración, de forma tal, de poder controlar todos los parámetros que requiera su comunicación.*

<sup>8</sup> Existe un archivo HOW-TO denominado Serial-HOWTO en Linux en donde podrá encontrar información relevante acerca de la comunicación serial y hasta incluso, cómo armar un cable serial para transmisión de datos entre equipos o bien para conectar una terminal Unix serial.