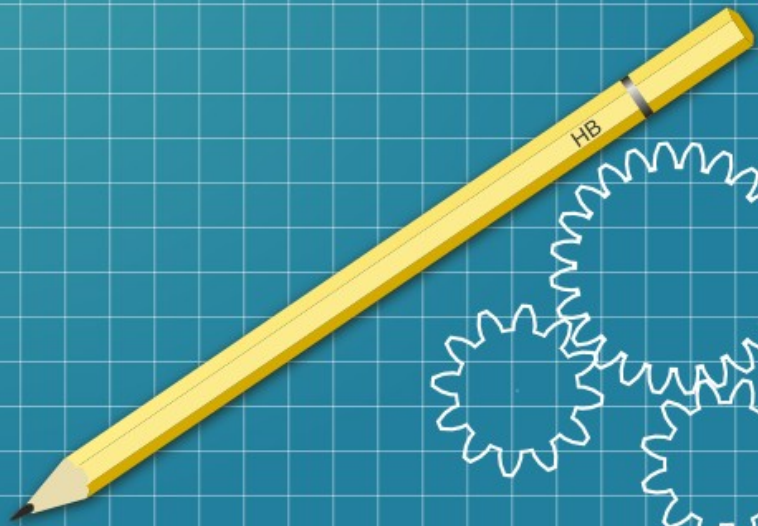
Programación de archivos a bajo nivel

```
#include<stdio.h>

int main(){
  int a = 10;
  int b;
  b ~= a;
  return 0;
}
```



Palomeque Nestor Levi

Manejo de archivos a bajo nivel

La programación a bajo nivel implica interactuar directamente con los componentes físicos de una computadora y el sistema operativo, utilizando funciones y características que están muy cerca del hardware. En C, un lenguaje de programación de nivel medio, puedes acceder a estas funcionalidades utilizando las funciones de bajo nivel proporcionadas por las bibliotecas estándar del lenguaje y del sistema operativo.

En las funciones usaremos los descriptores de archivo que son son números enteros que se pueden utilizar para manejar archivos o dispositivos.

Todo programa tiene abierto tres descriptores de archivos:

- 0 ó STDIN_FILENO: entrada estándar (stdin)
- 1 ó STDOUT_FILENO: salida estándar (stdout)
- 2 ó STDERR_FILENO: error estándar (stderr)

Función open()

La función **open()** se usa para abrir o crear un archivo en el sistema de archivos. Devuelve un descriptor de archivo, que es un entero que identifica de manera única al archivo abierto. Si hay un error, devuelve -1. Este descriptor de archivo se usa en otras funciones para leer, escribir, y realizar otras operaciones con el archivo.

```
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

pathname: La ruta al archivo que se abrirá o creará.

flags: Los flags que controlan el comportamiento de la apertura del archivo.

mode: Los permisos para el archivo si se está creando.

Valor de retorno: Devuelve un descriptor de archivo, número entero.

Algunos flags son:

- O_RDONLY: Abre el archivo en modo solo lectura. Con referencia al puerto serie utiliza este flag si solo necesitas leer datos.
- O_WRONLY: Abre el archivo en modo solo escritura. Con referencia al puerto serie utiliza este flag si solo necesitas escribir datos.
- O_RDWR: Abre el archivo en modo lectura y escritura. Comúnmente usado para puertos serie en los que necesitas tanto leer como escribir datos.
- O_APPEND: Escribe al final del archivo si se realizan escrituras.
- O_CREAT: Crea el archivo si no existe.
- O_TRUNC: Trunca el archivo a cero longitud después de abrirlo. Es decir, si el archivo ya existe, todos sus contenidos se eliminarán, dejándolo vacío.
- O_SYNC: Asegura que cada operación de escritura se complete físicamente en el dispositivo antes de que la llamada a write() se devuelva.
- O_EXCL: Si se combina con O_CREAT, asegura que la llamada a open() falle si el archivo ya existe.
- O_NOCTTY: En sistemas UNIX y Linux, cada proceso puede tener un terminal de control, que es el dispositivo de entrada/salida asociado al proceso. Este flag evita que el archivo se convierta en el terminal de control del proceso que lo abre. Esto es importante cuando se trabaja con **puerto serie**, así no tomará el control de terminal del proceso, evitando que el proceso reciba señales de control.
- O_NDELAY / O_NONBLOCK: Este flag también configura el archivo para operar en modo no bloqueante. Las operaciones de lectura en el puerto no bloquearán el proceso si el dispositivo no está listo. Esto significa que read() no esperará indefinidamente a que lleguen datos, y write() no esperará a que el dispositivo esté listo para aceptar datos.

Los permisos del archivo controlan quién puede leer, escribir y ejecutar el archivo, y se especifican utilizando un código octal. Cada dígito en el código octal representa un conjunto diferente de permisos: el primero para el propietario del archivo, el segundo para el grupo al que pertenece el archivo y el tercero para todos los demás usuarios. Los permisos disponibles son:

4 (r): Permisos de lectura. Permite al usuario leer el contenido del archivo.

2 (w): Permisos de escritura. Permite al usuario escribir o modificar el contenido del archivo.

1 (x): Permisos de ejecución. Permite al usuario ejecutar el archivo si es un programa ejecutable.

0 (-): Sin permisos. Deniega al usuario el acceso al archivo.

Los permisos se pueden combinar sumando los valores correspondientes. Por ejemplo, 0644 (el ‘0’ adelante indica notación Octal) especifica que el propietario tiene permisos de lectura y escritura (4 + 2 = 6), mientras que los demás usuarios solo tienen permiso de lectura (4).

Función close()

La función **close()** en C se utiliza para cerrar un archivo o descriptor de archivo abierto previamente. Cuando se cierra un archivo, se liberan los recursos asociados con él y se asegura que los datos en el búfer de salida se escriban en el archivo. Es importante cerrar los archivos después de usarlos para evitar problemas de pérdida de datos o recursos.

La función close() toma un argumento, que es el descriptor de archivo del archivo que se va a cerrar. Después de que un archivo se cierra con éxito, el descriptor de archivo ya no se puede utilizar para acceder al archivo. La función close() devuelve 0 si tiene éxito al cerrar el archivo y -1 si hay un error.

```c
#include<fcntl.h>  //File control definitions. Para open()
#include<unistd.h> // UNIX standard function definitions. Para close(), read() y write()
#include <stdio.h>   // Para perror(), se utiliza para imprimir un mensaje de error relacionado con las llamadas al sistema y las funciones de biblioteca estándar que fallan.
int main() {
    int fd;
    fd = open("archivo.txt", O_WRONLY | O_CREAT, 0644);  // Abrir un archivo en modo de escritura
    if (fd == -1) {
        perror("Error al abrir el archivo");
        return 1;
    }
    // .......Realizar operaciones de escritura en el archivo.......

    if (close(fd) == -1) {                  // Cerrar el archivo después de usarlo
        perror("Error al cerrar el archivo");
        return 1;
     }  else printf("El archivo se cerró correctamente.\n");

    return 0;
}
```

En este ejemplo, se abre el archivo "archivo.txt" en modo de escritura utilizando open(), se realizan operaciones de escritura en el archivo y luego se cierra el archivo utilizando close().

Función write()

La función **write()** se utiliza para escribir datos en un archivo o descriptor de archivo abierto. Se puede usar para escribir datos en un archivo regular, en una tubería, en un socket, o en cualquier otro tipo de archivo que soporte escritura.

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

fd: El descriptor de archivo donde se va a escribir.

buf: Un puntero al buffer que contiene los datos que se desean escribir.

count: El número de bytes a escribir desde el buffer.

Valor de retorno: la función **write()** devuelve el número de bytes que se han escrito exitosamente. Si hay un error, devuelve -1.

Nota:

- **const void*** es un tipo de puntero genérico constante. **void*** se utiliza para apuntar a cualquier tipo de datos, ya que no tiene un tipo asociado. Al añadir **const**, se indica que los datos apuntados no deben ser modificados a través de este puntero. En el contexto de **write()**, **const void* buf** significa que **write()** tomará un puntero a cualquier tipo de datos, y no modificará los datos apuntados.
- **size_t** es un tipo de dato que se utiliza para representar tamaños y conteos. Suele ser un **unsigned int** o un tipo similar que tiene un tamaño de 32 bits (4 bytes). En Sistemas de 64 Bits: **size_t** suele ser un **unsigned long** o un tipo similar que tiene un tamaño de 64 bits (8 bytes). Así sería la definición:

  ```
  typedef unsigned long size_t;
  ```

  La palabra clave **typedef** se usa para crear un alias para un tipo de dato
- **ssize_t** es un tipo de dato utilizado para representar tamaños y conteos de bytes, pero a diferencia de size_t, ssize_t es un tipo entero con signo. Esto significa que puede representar tanto valores positivos como negativos, lo que es útil para operaciones en las que la función puede indicar un error mediante un valor negativo.

Ejemplo de escritura

El siguiente ejemplo crea y escribe en un archivo el texto “Hola mundo!”

```c
#include <fcntl.h>   // Para open()
#include <unistd.h>  // Para write(), close()
#include <stdio.h>   // Para perror()
int main() {
    const char *mensaje = "Hola, mundo!";      // Datos a escribir

    int fd = open("archivo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);   // Abrir el archivo para escritura (crear si no existe, truncar si ya existe)
    if (fd == -1) {
        perror("Error al abrir el archivo");
        return 1;
    }
    // Escribir los datos en el archivo
    ssize_t bytes_escritos = write(fd, mensaje, 13);          //Se puede usar "strlen(mensaje)" para obtener el número de bytes sin incluir el final de cadena.
    if (bytes_escritos == -1) {
        perror("Error al escribir en el archivo");
        close(fd);
        return 1;
    }
    if (close(fd) == -1) {  // Cerrar el archivo
        perror("Error al cerrar el archivo");
        return 1;  }
    return 0;
}
```

Función read()

La función **read()** se utiliza para leer datos desde un archivo o dispositivo.

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- **fd**: Descriptor de archivo que se obtiene al abrir un archivo o dispositivo.

  buf: Puntero a un buffer (área de memoria) donde se almacenarán los datos leídos. Debe ser suficientemente grande para contener los datos que se intentan leer.

  count: Número máximo de bytes que se intentarán leer desde el descriptor de archivo.

  Valor de retorno: devuelve el número de bytes leídos, que puede ser menor que **count** si se llega al final del archivo o si hay menos datos disponibles. Si hay un error devuelve -1.

Ejemplo de lectura

El siguiente ejemplo abre y lee el archivo creado anteriormente

```c
#include <fcntl.h>   // Para open()
#include <unistd.h>  // Para read() y close()
#include <stdio.h>   // Para perror()
#define BUFFER_SIZE 100
int main() {
    char buffer[BUFFER_SIZE];    // Crear un buffer para almacenar los datos leídos

    int fd = open("archivo.txt", O_RDONLY); // Abrir el archivo para lectura
    if (fd == -1) {
        perror("Error al abrir el archivo");
        return 1;   }

    ssize_t bytes_leidos = read(fd, buffer, sizeof(buffer) - 1);  // Leer datos del archivo. Reservar espacio para el carácter nulo restando 1.
    if (bytes_leidos == -1) {
        perror("Error al leer el archivo");
        close(fd);
        return 1;   }

     buffer[bytes_leidos] = '\0'; // Añadir un carácter nulo para tratar el buffer como una cadena C

    printf("Datos leídos: %s\n", buffer);    // Imprimir los datos leídos

    if (close(fd) == -1) {  // Cerrar el archivo
        perror("Error al cerrar el archivo");
        return 1;  }
    return 0;
}
```

Ejemplos empleando puerto serie

A continuación realizaremos dos ejemplo para transmitir y recibir una cadena empleando el puerto serie y una placa arduino. El primero utilizará los parámetros por defectos del puerto y en el segundo ejemplo utilizaremos la biblioteca **termios** para establecer los parámetro del puerto serie desde nuestro programa.

- Primero al conectar la placa analizaremos que puerto se ha asignado a la misma. Para ellos debemos utilizar el comando: `sudo dmesg`
- A continuación utilizaremos el siguiente comando para visualizar la configuración por defecto del puerto correspondiente:
- `stty -F /dev/ttyUSB0 -a`
- Luego debemos modificar el archivo main.c para especificar dicho puerto.

Ejemplo 1 - Archivo mainDefault.c

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>

int main()
{
    int fd; // Descriptor de archivo para el puerto serie
    char buffer[256]={'\0'}; // Buffer para leer datos del puerto serie
    const char *mensaje = "Hola desde Linux\n"; // Mensaje a enviar. Importanto usar '\n' para que Arduino detecte el fin del string

    fd = open("/dev/ttyUSB0", O_RDWR | O_NOCTTY | O_NONBLOCK);  // Abrir el puerto serie en modo lectura y escritura, no como terminal de control y función read() "NO bloqueante".
    if (fd == -1) {
        perror("Error al abrir el puerto serie");
        return 1;     }

    sleep(2); //Cada vez que se abre el puerto, se resetea el ARDUINO por RTS, con esto se espera antes de escribir.
    printf("Escribiendo mensaje por el puerto serie: %s\n", mensaje);
    ssize_t bytes_escritos = write(fd, mensaje, strlen(mensaje));      // Escribir el mensaje en el puerto serie
    if (bytes_escritos == -1) {
        perror("Error al escribir en el puerto serie");
        close(fd);
        Return 1;    }


    //Limpia el búfer de entrada antes de empezar
    tcflush(fd, TCIFLUSH);

    sleep(2); //Como el arduino demora en responder, debo esperar un tiempo acorde. Si se usa O_NONBLOCK, esto es necesario
    printf("Leyendo mensaje por el puerto serie\n");
    ssize_t bytes_leidos = read(fd, buffer, sizeof(buffer) - 1);           // Leer la respuesta del puerto serie
    if (bytes_leidos == -1) {
        perror("Error al leer del puerto serie");
        close(fd);
        return 1;   }

    buffer[bytes_leidos] = '\0'; // Añadir un carácter nulo para tratar el buffer como una cadena C
    printf("Datos recibidos: %s\n", buffer);      // Mostrar los datos recibidos
    close(fd);  // Cerrar el puerto serie
    return 0;
}
```

Ejemplo 1 y 2 - programaArduino.ino

```
void setup() {
  Serial.begin(9600);    // Modificar la velocidad según corresponda
}

void loop() {
  if(Serial.available())
  {
    Serial.print("Hola desde Arduino. El mensaje recibido es: " + Serial.readStringUntil('\n'));
  }
}
```

El archivo programaArduino.ino es útil para ambos ejemplos.

- Modificar el programa de arduino para utilizar la velocidad por defecto del sistema operativo.
- Compilar el archivo mainDefault.c
- Ejecutar el programa.

Ejemplo 2 - termset.h

```c
#ifndef TERMSET_H
#define TERMSET_H
#include <termios.h>
#include <fcntl.h>

/**
 * @brief Función que configura los parámetros del puerto serie, para 8N1.
 *
 * @param fd Descriptor de archivo del puerto serie (ejemplo: /dev/ttyUSB0)
 * @param baudrate Velocidad de comunicación (ejemplo: 9600, 115200)
 * @param ttyold Estructura termios que contiene la configuración actual del puerto serie
 * @param ttynew Estructura termios que contiene la nueva configuración para aplicar
 * @return 0 si la configuración fue exitosa, -1 en caso de error
 */
int termset(int fd, int baudrate, struct termios *ttyold, struct termios *ttynew);
#endif // TERMSET_H
```

Para el ejemplo 2, estableceremos la velocidad del puerto desde nuestro programa a 115200 bps. Para ello utilizaremos el módulo **termset** que emplea la biblioteca **termios**

- Modificar el programa de arduino para utilizar la velocidad indicada.
- Modificar el archivo main.c con la misma velocidad
- Compilar el programa
- Ejecutarlo.

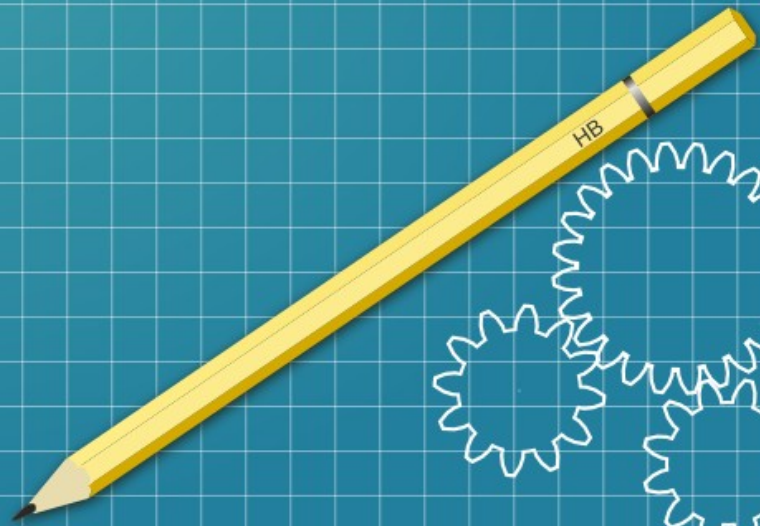
Ejemplo 2 - Archivo termset.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <termios.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include "termset.h"

// Función para configurar los parámetros del puerto serie
int termset(int fd, int baudrate, struct termios *ttyold, struct termios *ttynew)
{
  // Convertir la velocidad de baudios a la constante adecuada
  switch(baudrate)
  {
    case 115200: baudrate = B115200; break; // Velocidad de baudios 115200
    case 57600:  baudrate = B57600;  break; // Velocidad de baudios 57600
    case 38400:  baudrate = B38400;  break; // Velocidad de baudios 38400
    case 19200:  baudrate = B19200;  break; // Velocidad de baudios 19200
    case 9600:   baudrate = B9600;   break; // Velocidad de baudios 9600
    default:     baudrate = B115200; break; // Valor por defecto si no se encuentra el baudio
  }
  // Configurar el descriptor de archivo para operaciones de entrada/salida
  fcntl(fd, F_SETFL, 0);

  // Obtener los parámetros actuales del terminal y almacenarlos en ttyold
  if(tcgetattr(fd, ttyold) != 0)
  {
    perror("tcgetattr"); // Usar perror para imprimir el error
    return -1;
  }
  // Copiar la configuración antigua a la nueva estructura
  *ttynew = *ttyold;

  // Establecer la velocidad de entrada y salida
  cfsetospeed(ttynew, baudrate);
  cfsetispeed(ttynew, baudrate);

  // Configurar el puerto:
  // 8 bits de datos (CS8), sin paridad (PARENB y PARODD desactivados), 1 bit de parada (CSTOPB desactivado)
  ttynew->c_cflag = (ttynew->c_cflag & ~CSIZE) | CS8;
  ttynew->c_cflag &= ~(PARENB | PARODD);
  ttynew->c_cflag &= ~CSTOPB;
  // Ignorar las líneas de estado del módem y habilitar la lectura
  ttynew->c_cflag |= (CLOCAL | CREAD);

  // Configuración de entrada:
  // Desactivar el procesamiento de entrada (sin conversión de CR a NL, sin control de flujo XON/XOFF, etc.)
  ttynew->c_iflag &= ~(IGNBRK | BRKINT | ICRNL | INLCR | PARMRK | INPCK | ISTRIP | IXON);

  // Configuración de salida:
  // Desactivar el procesamiento de salida (sin conversión de NL a CR-NL, sin supresión de caracteres de relleno, etc.)
  ttynew->c_oflag &= ~(OCRNL | ONLCR | ONLRET | ONOCR | OFILL | OPOST);

  // Configuración de control de línea:
  // Desactivar el procesamiento de línea (sin eco, sin modo canónico, sin extensión de entrada, sin caracteres de señalización)
  ttynew->c_lflag &= ~(ECHO | ECHONL | ICANON | IEXTEN | ISIG);

  // Configurar el tiempo de espera y el número mínimo de caracteres para la lectura
  ttynew->c_cc[VMIN]  = 0;  // Esperar al menos 1 byte para la lectura
  ttynew->c_cc[VTIME] = 5;  // Tiempo de espera 0.5 segundos, luego del último dato recibido (timeout)

  if(tcsetattr(fd, TCSAFLUSH, ttynew) != 0)  // Aplicar los cambios inmediatamente y vacía los buffer I/O
  {
    perror("tcsetattr"); // Imprimir mensaje de error si tcsetattr falla
    return -1;
  }

  return 0; // Retornar 0 en caso de éxito
}
```

Ejemplo 2 -Archivo main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include "termset.h"
int main()
{
    int fd; // Descriptor de archivo para el puerto serie
    struct termios ttyold, ttynew; // Estructuras para configurar el puerto serie
    char buffer[256]; // Buffer para leer datos del puerto serie
    const char *mensaje = "Hola desde Linux\n"; // Mensaje a enviar. Importante usar '\n' para que Arduino detecte el fin de la cadena

// Abrir el puerto serie en modo lectura y escritura, no como terminal de control y no bloqueante
    fd = open("/dev/ttyUSB0", O_RDWR | O_NOCTTY | O_NONBLOCK);
    if (fd == -1) {
        perror("Error al abrir el puerto serie");
        return 1;
    }
 //Limpia el búfer de entrada antes de empezar
    tcflush(fd, TCIFLUSH);

// Configurar el puerto serie con la velocidad deseada
    if (termset(fd, 115200, &ttyold, &ttynew) != 0) {
        close(fd);
        return 1;
    }
    sleep(2); //Espero un tiempo antes de empezar a escribir. Al abrir el puerto suele resetear el Arduino.
   // Escribir el mensaje en el puerto serie
    printf("Escribiendo mensaje por el puerto serie: %s\n", mensaje);
    ssize_t bytes_escritos = write(fd, mensaje, strlen(mensaje));
    if (bytes_escritos == -1) {
        perror("Error al escribir en el puerto serie");
        close(fd);
        return 1;
    }
    sleep(2); //Como el arduino demora en responder, debo esperar un tiempo acorde.
    printf("Leyendo mensaje por el puerto serie\n");
    ssize_t bytes_leidos = read(fd, buffer, sizeof(buffer) - 1);  // Leer la respuesta del puerto serie
    if (bytes_leidos == -1) {
        perror("Error al leer del puerto serie");
        close(fd);
        return 1;
    }
    // Añadir un terminador nulo al final del buffer para asegurar que sea una cadena válida
    buffer[bytes_leidos] = '\0';
    printf("Datos recibidos: %s\n", buffer);      // Mostrar los datos recibidos

    if (tcsetattr(fd, TCSANOW, &ttyold) != 0)  perror("Error al restaurar la configuración del puerto serie");    // Restaurar la configuración original del puerto serie

    close(fd);  // Cerrar el puerto serie
    return 0;
}
```





HB