Serial Programming/termios

Contents

Introduction

termios is the newer (now already a few decades old) Unix API for terminal I/O. The anatomy of a program performing serial I/O with the help of termios is as follows:

- Open serial device with standard Unix system call **open**(2)
- Configure communication parameters and other interface properties (line discipline, etc.) with the help of specific termios functions and data structures.
- Use standard Unix system calls **read**(2) and **write**(2) for reading from, and writing to the serial interface. Related system calls like **readv**(2) and **writev**(2) can be used, too. Multiple I/O techniques, like blocking, non-blocking, asynchronous I/O (**select**(2) or **poll**(2), or signal-driven I/O (`SIGIO` signal)) are also possible. The selection of the I/O technique is an important part of the application's design. The serial I/O needs to work well with other kinds of I/O performed by the application, like networking, and must not waste CPU cycles.
- Close device with the standard Unix system call **close**(2) when done.

An important part when starting a program for serial I/I is to decide on the I/O technique to deploy.

The necessary declarations and constants for termios can be found in the header file `<termios.h>`. So code for serial or terminal I/O will usually start with

```
#include <termios.h>
```

Some additional functions and declarations can also be found in the `<stdio.h>`, `<fcntl.h>, and <unistd.h> header files.`

The termios I/O API supports two different modes: *doesn't old termio do this too? if yes, move paragraphs up to the general section about serial and terminal I/O in Unix).*

1. Canonical mode.
This is most useful when dealing with real terminals, or devices that provide line-by-line communication. The terminal driver returns data line-by-line.

2. Non-canonical mode.
In this mode, no special processing is done, and the terminal driver returns individual characters.

On BSD-like systems, there are three modes:

1. Cooked Mode.
Input is assembled into lines and special characters are processed.

2. Raw mode.
Input is not assembled into lines and special characters are not processed.

3. Cbreak mode.
Input is not assembled into lines but some special characters are processed.

Unless set otherwise, canonical (or cooked mode under BSD) is the default. The special characters processed in the corresponding modes are control characters, such as end-of-line or backspace. The full list for a particular Unix flavor can be found in the corresponding termios *man page*. For serial communication it is often advisable to use non-canonical, (raw or cbreak mode under BSD) to ensure that transmitted data is not interpreted by the terminal driver. Therefore, when setting up the communication parameters, the device should also configured for raw/non-canonical mode by setting/clearing the corresponding termios flags. It is also possible to enable or disable the processing of the special characters on an individual basis.

This configuration is done by using the `struct termios` data structure, defined in the `termios.h` header. This structure is central to both the configuration of a serial device and querying its setup. It contains a minimum of the following fields:

```
struct termios {
  tcflag_t c_iflag;    /* input specific flags (bitmask) */
  tcflag_t c_oflag;    /* output specific flags (bitmask) */
  tcflag_t c_cflag;    /* control flags (bitmask) */
  tcflag_t c_lflag;    /* local flags (bitmask) */
  cc_t     c_cc[NCCS]; /* special characters */
};
```

It should be noted that real `struct termios` declarations are often much more complicated. This stems from the fact that Unix vendors implement termios so that it is backward compatible with termio and integrate termio and termios behavior in the same data structure so they can avoid to have to implement the same code twice. In such a case, an application programmer may be able to intermix termio and termios code.

There are more than 45 different flags that can be set (via `tcsetattr()`) or got (via `tcgetattr()`) with the help of the `struct termios`. The large number of flags, and their sometimes esoteric and pathologic meaning and behavior, is one of the reasons why serial programming under Unix can be hard. In the device configuration, one must be careful not to make a mistake.

Opening/Closing a Serial Device

open(2)

A few decisions have to be made when opening a serial device. Should the device be opened for reading only, writing only, or both reading and writing? Should the device be opened for blocking or non-blocking I/O (non-blocking is recommended)? While **open**(2) can be called with quite a number of different flags controlling these and other properties, the following as a typical example:

```
const char *device = "/dev/ttyS0";
fd = open(device, O_RDWR | O_NOCTTY | O_NDELAY);
if(fd == -1) {
  printf( "failed to open port\n" );
}
```

Where:

device
: The path to the serial port (e.g. /dev/ttyS0)

fd
: The returned file handle for the device. -1 if an error occurred

O_RDWR
: Opens the port for reading and writing

O_NOCTTY
: The port never becomes the controlling terminal of the process.

O_NDELAY
: Use non-blocking I/O. On some systems this also means the RS232 DCD signal line is ignored.

NB: Be sure to #include <fcntl.h> as well for the constants listed above.

close(2)

Given an open file handle *fd* you can close it with the following system call

```
close(fd);
```

Basic Configuration of a Serial Interface

After a serial device has been opened, it is typical that its default configuration, like baud rate or line discipline needs to be overwritten with the desired parameters. This is done with a rather complex data structure, and the **tcgetattr**(3) and **tcsetattr**(3) functions. Before that is done, however, it is a good idea to check if the opened device is indeed a serial device (aka *tty*).

The following is an example of such a configuration. The details are explained later in this module.

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

struct termios  config;

//
// Check if the file descriptor is pointing to a TTY device or not.
//
if(!isatty(fd)) { ... error handling ... }

//
// Get the current configuration of the serial interface
//
if(tcgetattr(fd, &config) < 0) { ... error handling ... }

//
// Input flags - Turn off input processing
//
// convert break to null byte, no CR to NL translation,
// no NL to CR translation, don't mark parity errors or breaks
```

```
// no input parity check, don't strip high bit off,
// no XON/XOFF software flow control
//
config.c_iflag &= ~(IGNBRK | BRKINT | ICRNL |
                    INLCR | PARMRK | INPCK | ISTRIP | IXON);

//
// Output flags - Turn off output processing
//
// no CR to NL translation, no NL to CR-NL translation,
// no NL to CR translation, no column 0 CR suppression,
// no Ctrl-D suppression, no fill characters, no case mapping,
// no local output processing
//
// config.c_oflag &= ~(OCRNL | ONLCR | ONLRET |
//                     ONOCR | ONOEOT| OFILL | OLCUC | OPOST);
config.c_oflag = 0;

//
// No line processing
//
// echo off, echo newline off, canonical mode off,
// extended input processing off, signal chars off
//
config.c_lflag &= ~(ECHO | ECHONL | ICANON | IEXTEN | ISIG);

//
// Turn off character processing
//
// clear current char size mask, no parity checking,
// no output processing, force 8 bit input
//
config.c_cflag &= ~(CSIZE | PARENB);
config.c_cflag |= CS8;

//
// One input byte is enough to return from read()
// Inter-character timer off
//
config.c_cc[VMIN]  = 1;
config.c_cc[VTIME] = 0;

//
// Communication speed (simple version, using the predefined
// constants)
//
if(cfsetispeed(&config, B9600) < 0 || cfsetospeed(&config, B9600) < 0) {
    ... error handling ...
}

//
// Finally, apply the configuration
//
if(tcsetattr(fd, TCSAFLUSH, &config) < 0) { ... error handling ... }
```

This code is definitely not a pretty sight. It only covers the most important flags of the more than 60 termios flags. The flags should be revised, depending on the actual application.

Line-Control Functions

termios contains a number of line-control functions. These allow a more fine-grained control over the serial line in certain special situations. They all work on a file descriptor *fildes*, returned by an **open**(2) call to open the serial device. In the case of an error, the detailed cause can be found in the global `errno` variable (see **errno**(2)).

tcdrain

```
#include <termios.h>
int tcdrain(int fildes);
```

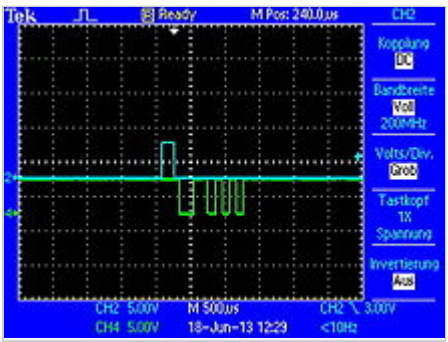
Wait until all data previously written to the serial line indicated by `fildes` has been sent. This means, the function will return when the UART's send buffer has cleared.
If successful, the function returns 0. Otherwise it returns -1, and the global variable `errno` contains the exact reason for the error.

use case

Todays computer are fast, have more cores and a lot of optimization inside. That can cause strange results because it result depends on circumstances the programmer may not aware of. See the example below:

```
set_rts();
write();
clr_rts();
```

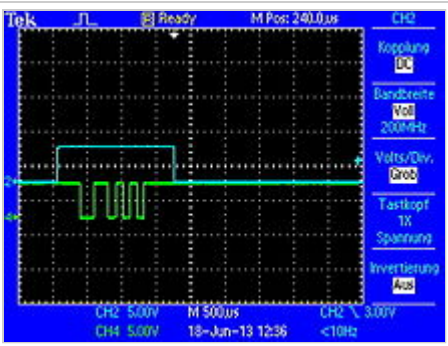
With that code you would expect a signal going up, the write and a signal going down. But that is wrong. Perhaps optimization causes the kernel to report success to write before the data are really written.



without tcdrain()

Now the same code using tcdrain()

```
set_rts();
write();
tcdrain();
clr_rts();
```



with tcdrain()

Suddenly the code behaves as expected, because the clr_rts(); is executed only when data really is written. Several programmers solve the problem by using sleep()/usleep() what may be not exactly what you want.

tcflow

```
#include <termios.h>
int tcflow(int fildes, int action);
```

This function suspends/restarts transmission and/or reception of data on the serial device indicated by *fildes*. The exact function is controlled by the *action* argument. *action* should be one of the following constants:

TCOOFF
: Suspend output.

TCOON
: Restarts previously suspended output.

TCIOFF
: Transmit a `STOP` (`xoff`) character. Remote devices are supposed to stop transmitting data if they receive this character. This requires the remote device on the other end of the serial line to support this software flow-control.

TCION
: Transmit a `START` (`xon`) character. Remote devices should restart transmitting data if they receive this character. This requires the remote device on the other end of the serial line to support this software flow-control.

If successful, the function returns 0. Otherwise it returns -1, and the global variable `errno` contains the exact reason for the error.

tcflush

```
#include <termios.h>
int tcflush(int fildes, int queue_selector);
```

Flushes (discards) not-sent data (data still in the UART send buffer) and/or flushes (discards) received data (data already in the UART receive buffer). The exact operation is defined by the *queue_selector* argument. The possible constants for *queue_selector* are:

TCIFLUSH
: Flush received, but unread data.

TCOFLUSH
: Flush written, but not send data.

TCIOFLUSH
: Flush both.

If successful, the function returns 0. Otherwise it returns -1, and the global variable `errno` contains the exact reason for the error.

tcsendbreak

```
#include <termios.h>
int tcsendbreak(int fildes, int duration_flag);
```

Sends a break for a certain duration. The *duration_flag* controls the duration of the break signal:

0
: Send a break of at least 0.25 seconds, and not more than 0.5 seconds.

any other value
: For other values than 0, the behavior is implementation defined. Some implementations interpret the value as some time specifications, others just let the function behave like *tcdrain()*.

A break is a deliberately generated framing (timing) error of the serial data – the signal's timing is violated by sending a series of zero bits, which also encompasses the start/stop bits, so the framing is explicitly gone.

If successful, the function returns 0. Otherwise it returns -1, and the global variable `errno` contains the exact reason for the error.

Reading and Setting Parameters

Introduction

There are more than 60 parameters a serial interface in Unix can have, assuming of course the underlying hardware supports every possible parameter - which many serial devices in professional workstations and Unix servers indeed do. This plethora of parameters and the resulting different interface configuration is what make serial programming in Unix and Linux challenging. Not only are there so many parameters, but their meanings are often rather unknown to contemporary hackers, because they originated at the dawn of computing, where things were done differently and are no longer known or taught in Little-Hacker School.

Nevertheless, most of the parameters of a serial interface in Unix are controlled via just two functions:

tcgetattr()
: For reading the current attributes.

and

tcsetattr()
: For setting serial interface attributes.

All information about the configuration of a serial interface is stored in an instance of the `struct termios` data type. **tcgetattr()** requires a pointer to a pre-allocated `struct termios` where it reads the values from. **tcsetattr()** requires a pointer to a pre-allocated and initialized `struct termios` where the function writes to.

Further, speed parameters are set via a separate set of functions:

cfgetispeed()
: Get line-in speed.

cfgetospeed()
: Get line-out speed.

cfsetispeed()
: Set line-in speed.

cfsetospeed()
: Set line-out speed.

The following sub-section explain the mentioned functions in more detail.

Attribute Changes

50+ attributes of a serial interface in Unix can be read with a single function: **tcgetattr()**. Among these parameters are all the option flags and, for example, information about which special character handling is applied. The signature of that function is as it follows:

```
#include <termios.h>
int tcgetattr(int fd, struct termios *attribs);
```

Where the arguments are:

fd
: A file handle pointing to an opened terminal device. The device has typically be opened via the **open**(2) system call. However, there are several more mechanisms in Unix to obtain a legitimate file handle (e.g. by inheriting it over a **fork**(2)/**exec**(2) combo). As long as the handle points to an opened terminal device things are fine.

***attribs**
: A pointer to a pre-allocated `struct termios`, where **tcgetattr()** will write to.

tcgetattr() returns an integer that either indicates success or failure in the way typical for Unix system calls:

0
: Indicates successful completion

-1
: Indicates failure. Further information about the problem can be found in the global (or thread-local) `errno` variable. See the **errno**(2), **intro**(2), and/or **perror**(3C) man page for information about the meaning of the `errno` values.

  Note; it is a typical beginner and hacker error to not check the return value and assume everything will always work.

The following is a simple example demonstrating the use of **tcgetattr()**. It assumes that standard input has been redirected to a terminal device:

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
int main(void) {
    struct termios attribs;
    speed_t speed;
    if(tcgetattr(STDIN_FILENO, &attribs) < 0) {
        perror("stdin");
        return EXIT_FAILURE;
    }
    /*
     * The following mess is to retrieve the input
     * speed from the returned data. The code is so messy,
     * because it has to take care of a historic change in
     * the usage of struct termios. Baud rates were once
     * represented by fixed constants, but later could also
     * be represented by a number. cfgetispeed() is a far
     * better alternative.
     */
    if(attribs.c_cflag & CIBAUDEXT) {
        speed = ((attribs.c_cflag & CIBAUD) >> IBSHIFT)
                 + (CIBAUD >> IBSHIFT) + 1;
    }
    else
    {
        speed = (attribs.c_cflag & CIBAUD) >> IBSHIFT;
    }
    printf("input speed: %ul\n", (unsigned long) speed);
    /*
     * Check if received carriage-return characters are
     * ignored, changed to new-lines, or passed on
     * unchanged.
     */
    if(attribs.c_iflag & IGNCR) {
        printf("Received CRs are ignored.\n");
    }
    else if(attribs.c_iflag & ICRNK)
    {
        printf("Received CRs are translated to NLs.\n");
    }
    else
    {
        printf("Received CRs are not changed.\n");
    }
    return EXIT_SUCCESS;
}
```

Once the above program is compiled and linked, let's say under the name `example`, it can be run as it follows:

```
./example < /dev/ttya
```

Assuming, `/dev/ttya` is a valid serial device. One can run **stty** to verify of the output is correct.

tcsetattr()

```
#include <termios.h>
tcsetattr( int fd, int optional_actions, const struct termios *options );
```

Sets the termios struct of the file handle *fd* from the options defined in *options*. *optional_actions* specifies when the change will happen:

TCSANOW
: the configuration is changed immediately.

TCSADRAIN

the configuration is changed after all the output written to *fd* has been transmitted. This prevents the change from corrupting in-transmission data.

TCSAFLUSH
same as above but any data received and not read will be discarded.

Baud-Rate Setting

Reading and setting the baud rates (the line speeds) can be done via the **tcgetattr**() and **tcsetattr**() function. This can be done by reading or writing the necessary data into the `struct termios`. However, this is a mess. The previous example for **tcgetattr**() demonstrates this.

Instead of accessing the data manually, it is highly recommended to use one of the following functions:

cfgetispeed()
Get line-in speed.

cfgetospeed()
Get line-out speed.

cfsetispeed()
Set line-in speed.

cfsetospeed()
Set line-out speed.

Which have the following signatures:

```
#include <termios.h>
speed_t cfgetispeed(const struct termios *attribs);
```

speed
The input baud rate.

attribs
The `struct termios` from which to extract the speed.

```
#include <termios.h>
speed_t cfgetospeed(const struct termios *attribs);
```

speed
The output baud rate.

attribs
The `struct termios` from which to extract the speed.

```
#include <termios.h>
int cfsetispeed(struct termios *attribs, speed_t speed);
```

attribs
The `struct termios` in which the input baud rate should be set.

speed
The input baud rate that should be set.

The function returns

0
If the speed could be set (encoded).

-1
If the speed could not be set (e.g. if it is not a valid or supported speed value).

```
#include <termios.h>
int cfsetospeed(struct termios *attribs, speed_t speed);
```

attribs
: The `struct termios` in which the output baud rate should be set.

speed
: The output baud rate that should be set.

The function returns

0
: If the speed could be set (encoded).

-1
: If the speed could not be set (e.g. if it is not a valid or supported speed value).

Here is a simple example for **cfgetispeed**(). **cfgetospeed**() works very similar:

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
int main(void) {
    struct termios attribs;
    speed_t speed;
    if(tcgetattr(STDIN_FILENO, &attribs) < 0) {
        perror("stdin");
        return EXIT_FAILURE;
    }
    speed = cfgetispeed(&attribs);
    printf("input speed: %lu\n", (unsigned long) speed);
    return EXIT_SUCCESS;
}
```

cfsetispeed() and **cfsetospeed**() work straight-forward, too. The following example sets the input speed of stdin to 9600 baud. Note, the setting will not be permanent, since the device might be reset at the end of the program:

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
int main(void) {
    struct termios attribs;
    speed_t speed;
    /*
     * Get the current settings. This saves us from
     * having to initialize a struct termios from
     * scratch.
     */
    if(tcgetattr(STDIN_FILENO, &attribs) < 0)
    {
        perror("stdin");
        return EXIT_FAILURE;
    }
    /*
     * Set the speed data in the structure
     */
    if(cfsetispeed(&attribs, B9600) < 0)
    {
        perror("invalid baud rate");
        return EXIT_FAILURE;
    }
    /*
     * Apply the settings.
     */
    if(tcsetattr(STDIN_FILENO, TCSANOW, &attribs) < 0)
    {
        perror("stdin");
        return EXIT_FAILURE;
    }
```

```
    /* data transmision should happen here */
    return EXIT_SUCCESS;
}
```

Modes

Special Input Characters

Canonical Mode

Everything is stored into a buffer and can be edited until a carriage return or line feed is entered. After the carriage return or line feed is pressed, the buffer is sent.

```
options.c_lflag |= ICANON;
```

where:

ICANON
: Enables canonical input mode

Non-Canonical Mode

This mode will handle a fixed number of characters and allows for a character timer.In this mode input is not assembled into lines and input processing does not occur.Here we have to set two parameters time and minimum number of characters to be received before read is satisfied and these are set by setting VTIME and VMIN characters for example if we have to set Minimum number of characters as 4 and we don't want to use any timer then we can do so as follows-:

```
options.c_cc[VTIME]=0;
options.c_cc[VMIN]=4;
```

Misc.

There are a few C functions that can be useful for terminal and serial I/O programming and are not part of the terminal I/O API. These are

```
#include <stdio.h>
char *ctermid(char *s);
```

This function returns the device name of the current controlling terminal of the process as a string (e.g. "/dev/tty01"). This is useful for programs who want to open that terminal device directly in order to communicate with it, even if the controlling terminal association is removed later (because, for example, the process forks/execs to become a daemon process).
**s* can either be `NULL` or should point to a character array of at least *L_ctermid* bytes (the constant is also defined in *stdio.h*). If **s* is `NULL`, then some internal static char array is used, otherwise the provided array is used. In both cases, a pointer to the first element of the char array is returned

```
#include <unistd.h>
int isatty(int fildes)
```

Checks if the provided file descriptor represents a terminal device. This can e.g. be used to figure out if a device will understand the commands send via the terminal I/O API.

```
#include <unistd.h>
char *ttyname (int fildes);
```

This function returns the device name of a terminal device represented by a file descriptor as a string.

```
#include <sys/ioctl.h>
ioctl(int fildes, TIOCGWINSZ, struct winsize *);
ioctl(int fildes, TIOCSWINSZ, struct winsize *);
```

These I/O controls allow to get and set the window size of a terminal emulation, e.g. an *xterm* in pixel and character sizes. Typically the get variant (TIOCGWINSZ) is used in combination with a SIGWINCH signal handler. The signal handler gets called when the size has changed (e.g. because the user has resized the terminal emulation window), and the application uses the I/O control to get the new size.

Example terminal program

A simple terminal program with termios.h can look like this:

Warning: In this program the VMIN and VTIME flags are ignored because the O_NONBLOCK flag is set.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h> // needed for memset

int main(int argc,char** argv)
{
        struct termios tio;
        struct termios stdio;
        int tty_fd;
        fd_set rdset;

        unsigned char c='D';

        printf("Please start with %s /dev/ttyS1 (for example)\n",argv[0]);
        memset(&stdio,0,sizeof(stdio));
        stdio.c_iflag=0;
        stdio.c_oflag=0;
        stdio.c_cflag=0;
        stdio.c_lflag=0;
        stdio.c_cc[VMIN]=1;
        stdio.c_cc[VTIME]=0;
        tcsetattr(STDOUT_FILENO,TCSANOW,&stdio);
        tcsetattr(STDOUT_FILENO,TCSAFLUSH,&stdio);
        fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);       // make the reads non-blocking



        memset(&tio,0,sizeof(tio));
        tio.c_iflag=0;
        tio.c_oflag=0;
        tio.c_cflag=CS8|CREAD|CLOCAL;           // 8n1, see termios.h for more information
        tio.c_lflag=0;
        tio.c_cc[VMIN]=1;
        tio.c_cc[VTIME]=5;

        tty_fd=open(argv[1], O_RDWR | O_NONBLOCK);
        cfsetospeed(&tio,B115200);            // 115200 baud
        cfsetispeed(&tio,B115200);            // 115200 baud

        tcsetattr(tty_fd,TCSANOW,&tio);
        while (c!='q')
        {
                if (read(tty_fd,&c,1)>0)        write(STDOUT_FILENO,&c,1);              // if new data is availa
                if (read(STDIN_FILENO,&c,1)>0)  write(tty_fd,&c,1);                     // if new data is availa
        }
```

```
        close(tty_fd);
}
```