# Serial Programming/Serial Linux

**Serial Programming**: Introduction and OSI Network Model -- RS-232 Wiring and Connections -- Typical RS232 Hardware Configuration -- 8250 UART -- DOS -- MAX232 Driver/Receiver Family -- TAPI Communications In Windows -- **Linux and Unix** -- Java -- Hayes-compatible Modems and AT Commands -- Universal Serial Bus (USB) -- Forming Data Packets -- Error Correction Methods -- Two Way Communication -- Packet Recovery Methods -- Serial Data Networks -- Practical Application Development -- IP Over Serial Connections

## Contents



# The Classic Unix C APIs for Serial Communication

## Introduction

## Scope

This page is about the classic Unix C APIs for controlling serial devices. Languages other than C might provide appropriate wrappers to these APIs which look similar, or come with their own abstraction (e.g. Java). Nevertheless, these APIs are the lowest level of abstraction one can find for serial I/O in Unix. And, in fact they are also the highest abstraction in C on standard Unix. Some Unix versions ship additional vendor-specific proprietary high-level APIs. These APIs are not discussed here.

Actual implementations of classic Unix serial APIs do vary in practice, due to the different versions of Unix and its clones, like Linux. Therefore, this module just provides a general outline. It is highly recommended that you study a particular Unix version's manual (man pages) when programming for a serial device in Unix. The relevant man pages are not too great a read, but they are usually complete in their listing of options and parameters. Together with this overview it should be possible to implement programs doing serial I/O under Unix.

## Basics

Linux, or any Unix, is a multi-user, multi-tasking operating system. As such, programs usually don't, and are usually not allowed to, access hardware resources like serial UARTs directly. Instead, the operating system provides

1. low-level drivers for mapping the device into the file system (`/dev` and/or `/device/` file system entries),
2. the standard system calls for opening, reading, writing, and closing the device, and
3. the standard system call for controlling a device, and/or
4. high-level C libraries for controlling the device.

The low-level driver not only maps the device into the file system with the help of the kernel, it also encapsulates the particular hardware. The user often does not even know or care what type of UART is in use.

Classic Unix systems often provide two different device nodes (or minor numbers) for serial I/O hardware. These provide access to the same physical device via two different names in the `/dev` hierarchy. Which node is used affects how certain serial control signals, such as DCD (data carrier detect), are handled when the device is opened. In some cases this can be changed programmatically, making the difference largely irrelevant. As a consequence, Linux only provides the different devices for legacy programs.

Device names in the file system can vary, even on the same Unix system, as they are simply aliases. The important parts of a device name (such as in `/dev`) are the major and minor numbers. The major number distinguishes a serial port, for example, from a keyboard driver, and is used to select the correct driver in the kernel. Note that the major number differs between different Unix systems. The minor number is interpreted by the device driver itself. For serial device drivers, it is typically used to detect which physical interface to use. Sometimes, the minor number will also be used by the device driver to determine the DCD behavior or the hardware flow control signals to be used.

The typical (but not standardized, see above) device names under Unix for serial interfaces are:

**/dev/tty*xxx***
: Normal, generic access to the device. Used for terminal and other serial communication (originally for **t**ele**ty**pes). More recently, they are also used in modem communication, for example, whereas the `/dev/cua`*xxx* was used on older systems.
See the following module on how terminal I/O and serial I/O relate on Unix.

**/dev/cua*xxx***

Legacy device driver with special DCD handling. Typically this was used for accessing a modem on old Unix systems, such as running the UUCP communication protocol over the serial line and the modem. The *cu* in the name stands for the [[#cu]] program. The *a* for ACU (automatic call unit).

The **xxx** part in the names above is typically a one or two digit number, or a lowercase letter, starting at 'a' for the first interface.

PC-based Unix systems often mimic the DOS/Windows naming for the devices and call them `/dev/com`*xxx*. Linux system generally call serial ports `/dev/ttyS`*xxx* instead.

To summarize, when programming for the serial interface of a Unix system it is **highly advisable** to provide complete configuration for the device name. Not even the typical `/dev` path should be hard coded.

Note, devices with the name `/dev/pty`*xxx* are pseudo terminal devices, typically used by a graphical user interface to provide a terminal emulator like *xterm* or *dtterm* with a "terminal" device, and to provide a terminal device for network logins. There is no serial hardware behind these device drivers.

## Serial I/O via Terminal I/O

### Basics

Serial I/O under Unix is implemented as part of the terminal I/O capabilities of Unix. And the terminal I/O capabilities of Unix were originally the typewriter/teletype capabilities. Terminal I/O is not limited to terminals, though. The terminal I/O API is used for communication with many serial devices other than terminals, such as modems and printers.

The terminal API itself has evolved over time. These days three terminal APIs are still used in Unix programs and can be found in recent Unix implementations. A fourth one, the very old one from Unix Version 6 exists, but is quite rare these days.

The three common ones are:

1. V7, 4BSD, XENIX style device-specific ioctl-based API,
2. An old one called termio
3. A newer one (although still already a few decades old), which is called termios (note the additional 's').

The newer `termios` API is based on the older `termio` API, and so the two `termio...` APIs share a lot of similarities. The `termios` API has also undergone changes since inception. For example, the method of specifying the baud rate has changed from using pre-defined constants to a more relaxed schema (the constants can still be used as well on most implementations).

Systems that support the newer `termios` often also support the older `termio` API, either by providing it in addition, or by providing a `termios` implementation with data structures which can be used in place of the `termio` data structures and work as `termio`. These systems also often just provide one man page under the older name **termio***(7)* which is then in fact the `termios` man page, too.

In addition, some systems provide other, similar APIs, either in addition or as a replacement. `termiox` is such an API, which is largely compatible with `termio` and adds some extensions to it taken from `termios`. So `termiox` can logically be seen as an intermediate step between `termio` and `termios`.

The terminal I/O APIs rely on the standard system calls for reading and writing data. They don't provide their own reading/writing functions. Reading and writing data is done via the **read***(2)* and **write***(2)* system calls. The terminal I/O APIs just add functions for controlling and configuring the device. Most of this happens via the **ioctl***(2)* system call.

Unfortunately, whichever of the standard APIs is used, one fact holds for all of them: They are a slight mess. Well, not really. Communication with terminals was and is a difficult issue, and the APIs reflect these difficulties. But due to the fact that one can do "everything" with the APIs, it is overwhelming when one "just" wants to do some serial communication. So why is there no separate serial-I/O-only API in Unix? There are probably two reasons for this:

1. Terminals/teletypes were the first, and apparently very important, serial devices which were connected to Unix. So that API was created first.
2. Once the API was there, there was no need to create a separate one for serial I/O only, since a large part of terminal I/O is serial I/O, and all needed features were already there in the terminal I/O API.

So which API should one use? There is one good reason to use the old V7 API. It is the simplest among the APIs - after going through some initialization woes on modern Unix systems. In general, however, the newer `termios` API makes the most sense, although it is the most complex one.

### Line Discipline

When programming serial interfaces on Unix, there is one phrase - *line discipline* - which can drive programmers crazy. The line discipline provides the hardware-independent interface for the communication between the computer and the terminal device. It handles such things as editing, job control, and special character interpretation, and performs transformations on the incoming and outgoing data.

This is useful for terminal communication (e.g. when a backspace character should erase the latest character from the send buffer before it goes over the wire, or when different end-of-line character sequences between the terminal and the computer need to be converted). These features are, however, hardly useful when communicating with the plethora of other serial devices, where unaltered data communication is desired.

Much of the serial programming in Unix is hitting the line discipline which is in use over the head so it doesn't touch the data. Monitoring what actually goes over the wire is a good idea.

## Unix V6/PWB

Unix *Bell Version 6* with the *programmer's workbench* (PWB) was released in 1975 to universities. It was the first Unix with an audience outside AT&T. It already had a terminal programming API. Actually, at that point it was the *typewriter* API. That API is not described here in depth.

The usage of this API can in theory be identified by the presence of the following signature in some source code:

```
#include <sgtty.h>
stty(fd, data)
int fd;
char *data;
gtty(fd, data)
int fd;
char *data;
```

In theory, because at that time the C language was still a little bit different.

`data` is supposed to point to a

```
struct {
   char ispeed, ospeed;
   char erase, kill;
   int  mode;
} *data;
```

structure. That structure later became `struct sgttyb` in Unix V7. Finding the V6 API in source code should be rare. Anyhow, recent Unix versions and clones typically don't support this API any more.

## Unix V7

See Serial Programming:Unix/V7

## termios

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

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

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

        unsigned char c='D';
        tcgetattr(STDOUT_FILENO,&old_stdio);

        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);              // i1
                if (read(STDIN_FILENO,&c,1)>0)  write(tty_fd,&c,1);                     // i1
```

```
        }

        close(tty_fd);
        tcsetattr(STDOUT_FILENO,TCSANOW,&old_stdio);

        return EXIT_SUCCESS;
}
```

See Serial_Programming:Unix/termios

### termio / ioctl(2)

See Serial Programming:Unix/termio

# Serial I/O on the Shell Command Line

## Introduction

It is possible to do serial I/O on the Unix command line. However, the available control is limited. Reading and writing data can be done with the shell I/O redirections like <, >, and |. Setting basic configuration, like the baud rate, can be done with the `stty` (set terminal type) command.

There is also `libserial` for Linux. It's a simple C++ class which hides some of the complexity of termios.

## Configuration with `stty`

The Unix command `stty` allows one to configure a "terminal". Since all serial I/O under Unix is done via terminal I/O, it should be no surprise that stty can also be used to configure serial lines. Indeed, the options and parameters which can be set via stty often have a 1:1 mapping to termio/termios. If the explanations regarding an option in the **stty***(1)* man page is not sufficient, looking up the option in the termio/termios man page can often help.

On "modern" (System V) Unix versions, stty changes the parameters of its current **standard input**. On older systems, stty changes the parameters of its current **standard output**. We assume a modern Unix is in use here. So, to change the settings of a particular serial interface, its device name must be provided to stty via an I/O redirect:

```
stty parameters < /dev/com0  # change setting of /dev/com0
```

On some systems, the settings done by stty are reverted to system defaults as soon as the device is closed again. This closing is done by the shell as soon as the `stty` *`parameters`* `< /dev/com0` command has finished. So when using the above command, the changes will only be in effect for a few milliseconds.

One way to keep the device open for the duration of the communication is to start the whole communication in a sub shell (using, for example, '( ... )'), and redirecting that input. So to send the string "ATI0" over the serial line, one could use:

```
( stty parameters
  echo "ATI0"
) < /dev/com0 > /dev/com0
```

Interweaving sending and receiving data is difficult from the command line. Two processes are needed; one reading from the device, and the other writing to the device. This makes it difficult to coordinate commands sent with the responses received. Some extensive shell scripting might be needed to manage this.

A common way to organize the two processes is to put the reading process in the background, and let the writing process continue to run in the foreground. For example, the following script configures the device and starts a background process for copying all received data from the serial device to standard output. Then it starts writing commands to the device:

```
# Set up device and read from it.
# Capture PID of background process so it is possible
# to terminate background process once writing is done
# TODO: Also set up a trap in case script is killed
#       or crashes.
( stty parameters; cat; )& < /dev/com0
bgPid=$?
# Read commands from user, send them to device
while read cmd; do
   echo "$cmd"
done >/dev/com0
# Terminate background read process
kill $bgPid
```

If there is a chance that a response to some command might never come, and if there is no other way to terminate the process, it is advisable to set up a timeout by using the alarm signal and `trap` that signal (signal 14), or simply kill the process:

```
trap timeout 14
timeout() {
   echo "timeout occurred"
}
pid=$$
( sleep 60 ; kill -14 $pid; )& # send alarm signal after 60 sec.
# normal script contents goes here
```

or

```
pid=$$
( sleep 60; kill -9 $pid;)& # brutally kill process after 60 sec.
# normal script contents goes here
```

# Permanent Configuration

### Overview

It is possible to provide a serial line with a default configuration. On classic Unix this is done with entries in the `/etc/ttytab` configuration file, on newer (System V R4) systems with `/etc/ttydefs`.

The default configurations make some sense when they are used for setting up terminal lines or dialup lines for a Unix system (and that's what they are for). However, such default configurations are not of much use when doing some serial communication with some other device. The correct function of the communication program should better not depend on some operating system configuration. Instead, the application should be self-contained and configure the device as needed by it.

### /etc/ttytab

The ttytab format varies from Unix to Unix, so checking the corresponding man page is a good idea. If the device is not intended for a terminal (no login), then the *getty* field (sometimes also called the program field, usually the 3rd field) for the device entry should be empty. The init field (often the 4th field) can contain an initialization command. Using `stty` here is a good idea. So, a typical entry for a serial line might look like:

```
# Device    TermType    Getty   Init
tty0        unknown     ""      "stty parameters"
```

### /etc/ttydefs

Just some hints:

`/etc/ttydefs` provides the configuration as used by the **ttymon** program. The settings are similar to the settings possible with stty.

`ttymon` is a program which is typically run under control of the Service Access Controller (SAC), as part of the Service Access Facility (SAF).

*TODO: Provide info to set up all the sac/sacadm junk.*

### /etc/serial.conf

Just some hints:

A Linux-specific way of configuring serial devices using the **setserial** program.

### tty

tty with the **-s** option can be used to test if a device is a terminal (supports the termio/termios ioctl()'s). Therefore it can also be used to check if a given file name is indeed a device name of a serial line.

```
echo "Enter serial device name: \c"
read dev
if tty -s < "$dev"; then
  echo "$dev is indeed a serial device."
else
  echo "$dev is not a serial device."
fi
```

### tip

It is a simple program for establishing a terminal connection with a remote system over a serial line. `tip` takes the necessary communication parameters, including the parameters for the serial communication, from a tip-specific configuration file. Details can be found in the `tip`*(1)* manual page.

Example:

To start the session over the first serial interface (here `ttya`):

```
tip -9600 /dev/ttya
```

To leave the session:

```
~.
```

### uucp

#### Overview

Uucp (Unix-to-Unix-Copy) is a set of programs for moving data over serial lines/modems between Unix computers. Before the rise of the Internet uucp was the heart and foundation of services like e-mail and Usenet (net news) between Unix computers. Today uucp is largely insignificant. However, it is still a good choice if two or more Unix systems should be connected via serial lines/modems.

The uucp suite also contains command line tools for login over a serial line (or another UUCP bearer to a remote system. These tools are `cu` and `ct`. They are e.g. useful when trying to access a device connected via a serial line and when debugging some serial line protocol.

#### cu

`cu` "call another UNIX system", does what the name implies. Only, that the other system does not have to be a UNIX system at all. It just sets up a serial connection, possibly by dialing via a modem.

`cu` is the oldest Unix program for serial communication. It's the reason why some serial devices on classic Unix systems are called something like `/dev/cul0` and `/dev/cua0`. Where *cu* of course stands for the `cu` program supposed to use the devices, *l* stands for *line* - the communication line, and *a* for acu (automatic call unit).

#### ct

`ct` is intended to spawn a login to a remote system over a modem line, serial line, or similar bearer. It uses the uucp devices list to find the necessary dialing (modem) commands, and the serial line settings.

## System Configuration

*inittab, ttytab, SAF configuration*

## Other Serial Programming Articles

**Serial Programming**: Introduction and OSI Network Model -- RS-232 Wiring and Connections -- Typical RS232 Hardware Configuration -- 8250 UART -- DOS -- MAX232 Driver/Receiver Family -- TAPI Communications In Windows -- **Linux and Unix** -- Java -- Hayes-compatible Modems and AT Commands -- Universal Serial Bus (USB) -- Forming Data Packets -- Error Correction Methods -- Two Way Communication -- Packet Recovery Methods -- Serial Data Networks -- Practical Application Development -- IP Over Serial Connections

# External links

- Serial Port Programming in Linux (http://trainingkits.gweb.io/serial-linux.html)
- pySerial (http://pyserial.sourceforge.net/) helps Python programmers use the serial port.