# 1. History of TTY - remote terminals are very old technology

In the 70s, computers were expensive. So many employees at an institution would share a single computer, and each person could have their own “terminal” to that computer.

1978. The Digital Equipment Corporation VT100 was the first ANSI X3.64 compliant terminal and featured lots of innovations including control by an Intel 8085 microprocessor (rather than custom discrete logic), scrolling regions, different character sizes (regular, compressed, double-wide, double-high-and-wide), smooth scrolling, and on-screen keyboard-driven setup rather than the knobs, DIP switches, jumpers, or screws used by earlier CRTs, including DEC's own [VT05](http://www.columbia.edu/cu/computinghistory/vt05.jpg), [VT50](http://www.columbia.edu/cu/computinghistory/vt50.jpg), and [VT52](http://www.columbia.edu/cu/computinghistory/vt52.html)**.**

The VT100 is by all measures the industry-standard terminal. and the VT100 is also the basis of the X-terminal (xterm) specification.

The VT100 series was followed by the VT200, VT300, VT400, and VT500 series, which add F-keys, an editing keypad, character-set switching, character-cell line-and-box drawing and math, true graphics (ReGIS, Sixel, Tektronix), multiple sessions, and so on.

# Terminal

**Once a program is loaded into memory to be executed, it becomes a process attached to the current terminal.**

## 2.1

what about Ctrl + another letter?

Next I got curious about – if I send Ctrl+e, what byte gets sent?

It turns out that it’s literally just the number of that letter in the alphabet, like this:

Ctrl+a => 1

Ctrl+b => 2

Ctrl+c => 3

Ctrl+d => 4

…

Ctrl+z => 26

Also, Ctrl+Shift+b does the exact same thing as Ctrl+b (it writes 0x2).

What about other keys on the keyboard? Here’s what they map to:

Tab -> 0x9 (same as Ctrl+I, since I is the 9th letter)

Escape -> \x1b

Backspace -> \x7f

Home -> \x1b[H

End: \x1b[F

Print Screen: \x1b\x5b\x31\x3b\x35\x41

Insert: \x1b\x5b\x32\x7e

Delete -> \x1b\x5b\x33\x7e

My Meta key does nothing at all

What about Alt? From my experimenting (and some Googling), it seems like Alt is literally the same as “Escape”, except that pressing Alt by itself doesn’t send any characters to the terminal and pressing Escape by itself does. So:

alt + d => \x1bd (and the same for every other letter)

alt + shift + d => \x1bD (and the same for every other letter)

etcetera

Let’s look at one more example!

## 2.2 changing window size

stty -F X rows Y // where X is the TTY device, and Y is half the terminal height.

## 2.3 回显 - Character Echoing

As you type, your terminal emulator transmits information to the kernel. Usually, the kernel echoes the same information back to the terminal emulator, allowing you to see what you type. Without character echoing, you can't see what you type, but we're in cooked mode so the line editing facilities are still working.

# 3 Others

In a naïve system, the UART driver would then deliver the incoming bytes directly to some application process. But such an approach would lack the following essential features:

**Line editing.** Most users make mistakes while typing, so a backspace key is often useful. The operating system provides an editing buffer and some rudimentary editing commands (backspace, erase word, clear line, reprint), which are enabled by default inside the line discipline. Advanced applications may disable these features by putting the line discipline in **raw** mode instead of the default cooked (or **canonical**) mode. The kernel provides several different line disciplines. Only one of them is attached to a given serial device at a time. The default discipline, which provides line editing, is called **N\_TTY(drivers/char/n\_tty.c)**

**Session management.** The user probably wants to run several programs simultaneously, and interact with them one at a time. If a program goes into an endless loop, the user may want to kill it or suspend it. Programs that are started in the background should be able to execute until they try to write to the terminal, at which point they should be suspended. Likewise, user input should be directed to the foreground program only. The operating system implements these features in the **TTY core(drivers/char/tty\_io.c)**.

To facilitate moving the terminal emulation into userland, while still keeping the TTY subsystem (session management and line discipline) intact, the pseudo terminal or pty was invented.

cat /proc/tty/ldiscs

# 4 Jobs and sessions

Job control is what happens when you press ^Z to suspend a program, or when you start a program in the background using &.

# 5 Signal madness

In UNIX, the kernel communicates with processes by sending paralyzing or deadly signals to them. Processes may intercept some of the signals, and try to adapt to the situation, but most of them don't..

You can use the command kill -l to see which signals your system implements. This is what it may look like:

suguoxu@sgx:~$ kill -l

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP

6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1

11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM

16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP

21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ

26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR

31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3

38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8

43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13

48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12

53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7

58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2

63) SIGRTMAX-1 64) SIGRTMAX

suguoxu@sgx:~$

Suppose that you are editing a file in your (terminal based) editor of choice. The cursor is somewhere in the middle of the screen, and the editor is busy executing some processor intensive task, such as a search and replace operation on a large file. Now you press ^Z. Since the line discipline has been configured to intercept this character (^Z is a single byte, with ASCII code 26), you don't have to wait for the editor to complete its task and start reading from the TTY device. Instead, the line discipline subsystem instantly sends SIGTSTP to the foreground process group. This process group contains the editor, as well as any child processes created by it.

# 6 Flow control and blocking I/O

# 7 Configuring the TTY device

# 8 tty 命令

stty

tty

The & causes the command to run as a background job.

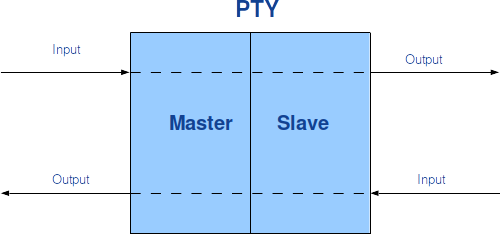
# 9 Why the TTY line discipline exists in the kernel

<https://utcc.utoronto.ca/~cks/space/blog/unix/TTYLineDisciplineWhy>

# 10 pseudo-terminals

<http://www.rkoucha.fr/tech_corner/pty_pdip.html>

A pseudo-terminal is a pair of character mode devices also called pty. One is master and the other is slave and they are connected with a bidirectional channel. Any data written on the slave side is forwarded to the output of the master side. Conversely, any data written on the master side is forwarded to the output of the slave side



The slave side behaves exactly as a standard terminal as any process can open it to make it its standard input and outputs. So, all the operations like disabling the echo, setting the line mode or canonical mode are available.

The master side is not a terminal. It is just a device which permits to send/receive data to/from the slave side.

In the Unix world, there are multiple implementations of the pseudo-terminals. There are the BSD and the System V versions. The Linux world recommends the system V implementation also called "Unix 98 pty". This is the one we are going to study below.

# 附录

## The DEC VT100 Terminal

