Terminals

A dumb terminal is basically a text display with a keyboard.

They typically have regular RS232 serial ports, displaying and sending ASCII characters

They were the main user interface to big mainframe computers, and minicomputers, used remotely via modems

Originally they were very dumb, they couldn't do much more than a printer. Early terminals **were** printers!

Companies eventually started making dumb terminals "smarter", with cursor positioning, colors, and graphics.

There now exists special sequences of characters for special functions, such as cursor keys, and control and escape sequences such as clearing the screen, moving the cursor, backspacing, and color

Different terminal types have different sequences. There is no standard.

Example VT100 escape sequences:

- move the cursor to row 12, column 53: ESC[12;53H
- clear the screen: ESC[HESC[J
- cursor up: ESC[A

UNIX systems now transparently support full-screen applications (emacs, top, etc) on hundreds of terminals, both real and emulated (e.g. putty)

Real dumb terminals are largely a thing of the past, instead we emulate their function with programs such as telnet

Most of these emulated terminals use escape sequences similar to the VT100, the closest thing to a "standard."

Handling different kinds of terminals

A terminal database contains definitions of terminal capabilities and corresponding control sequences (clearing the screen, moving the cursor, backspacing, color, etc).

Older programs used the **termcap** database (**/etc/termcap**) and library. Considered obsolete.

Newer programs use the **terminfo** terminal capability database and libraries. Every terminal is described in a separate binary file, typically under the **/lib/terminfo** or **/usr/share/terminfo** directories. The *tic* program can be used to generate database files for new terminal types that then stored in **/etc/terminfo**.

The **TERM** environment variable is used to indicate the type of terminal being used.

The **terminal driver** provides additional capabilities to those of the terminal. We interact with the terminal driver through a /dev/XXXXX device (pts, tty3, ttyS0, etc). The **/dev/tty** device always refers to the current terminal.

Canonical Mode

By default, the terminal driver is in standard or **canonical** mode and **read** will block until ENTER is pressed. Backspaces are handled by the driver and do not reach the program.

The terminal driver has many other features. It translates interrupt characters (CTRL-C, CTRL-Z) to signals, echoes characters, does CR to NL translation, allows raw input, etc.

Canonical mode is also called **cooked** mode (vs. raw mode)

Non-Canonical Mode

- In **non-canonical** or **raw** mode we have much greater control over input processing and can read a single character at a time.
- The **termios** routines (in **termios.h**) are used to get and set **terminal driver** modes (or attributes).
- You can change input, output, control, and local attributes. (See **man termios**)

termios - set terminal characteristics

"termios" controls behavior of the Unix terminal driver, including:

- echoing on/off
- input buffered until newline, or able to read each character immediately
- interpretation of backspace: edit or read directly
- character conversions (CR/LF)

In C programs, you use a special system structure called *termios*, with the functions *tcgetattr* and *tcsetattr*

In shell scripts you use the "stty" command

- sty - will show an abbreviated list of terminal capabilities
- stty -a - will show all terminal capabilities
- stty -g - will show all terminal capabilities in stty-readable form
- stty -echo - disables echoing of typed characters
- stty -icanon min **nnn** time **ttt** - turns off canonical mode, doesn't require "enter" to read
    - *nnn* indicates number of characters accepted
    - *ttt* indicates time allowed between each character, in 10th's of seconds
- stty -icrnl - disables carriage-return to newline conversion
- stty echo icanon icrnl - enables these capabilities
- if you get into problems with terminal control, try typing "stty sane" or "control-j stty sane control-j"

man termios and *man stty*

Using termios in a Shell Script

A simple "password" script:

```
printf "Enter password: "
stty -echo
read password
stty echo
printf "\nYou entered $password\n\n"
exit 0
```

To read one character, including cursor keys:

```
oldsettings=$(stty -g)
stty -icanon min 0 time 0 -icrnl
key=
while [ "$key" -ne "q" ]; do
      printf "Hit any key: "
      key=
      while [ "$key" -eq "" ]; do
            read key
      done
      if [ "$key" -eq "^M" ]; then
            key="Enter"
      elif [ "$key" -eq "^[[A" ]; then
            key="Up"
      elif [ "$key" -eq "^[[B" ]; then
            key="Down"
      elif [ "$key" -eq "^[[D" ]; then
            key="Left"
      elif [ "$key" -eq "^[[C" ]; then
            key="Right"
      fi
      printf "\nYou hit the \"$key\" key\n"
done
stty $oldsettings
```

Is stdout a Terminal?

Use the *isatty(fd)* routine to determine if file descriptor *fd* is a terminal (tty device).

open and *read*/*write* the */dev/tty* device to communicate directly with the terminal even if stdin or stdout have been redirected.

In a shell script redirect *read*/*echo* from/to /dev/tty to communicate with the terminal regardless of stdin/stdout.

The termios Routines

- Use ***tcgetattr*()** to get current attributes:

```
int tcgetattr(int fd, struct termios *t);
```

- Use ***tcsetattr*()** to set attributes:

```
int tcgetattr(int fd, int act, struct termios *t);
```

- **act** controls when the changes are applied. It may be TCSANOW (apply now), TCSADRAIN (apply when output is complete), or TCSAFLUSH (apply when output is complete, but flushes waiting input).
- The *termios* routines use a *termios* struct to get and set modes:

```
#include <termios.h>
struct termios {
      tcflag_t c_iflag;       /* input modes */
      tcflag_t c_oflag;       /* output modes */
      tcflag_t c_cflag;       /* control modes */
      tcflag_t c_lflag;       /* local modes */
      cc_t  c_cc[NCCS];       /* control chars */
};
```

- Input and output modes control how input and output are processed. Control modes specify hardware characteristics (parity, stop bits, etc.). Local modes can turn on/off raw mode, echoing, and signal processing.
- Attributes are set or cleared by logically ORing constants defined in **termios.h**.
- The control characters array of the ***termios*** struct allows us to control which characters cause the terminal driver to generate INT and QUIT signals and which characters are used for ERASE, KILL, EOF, etc. These are typically ^C, ^\, ^H, ^K, ^D.

The TIME and MIN elements of the array are used in RAW mode and determine when characters are returned by a **read()**.

Using termios in a C Program

An example of a password program in C:

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

#define PASSWORD_LEN 8

int main(void)
{
	struct termios initial_settings, new_settings;
	char password[PASSWORD_LEN + 1];
	tcgetattr(fileno(stdin), &initial_settings);
	new_settings = initial_settings;
	new_settings.c_lflag &= ~ECHO;
	printf("Enter password: ");

	if (tcsetattr(fileno(stdin), TCSAFLUSH, &new_settings) != 0)
		fprintf(stderr, "Could not set attributes\n");
	else {
		fgets(password, PASSWORD_LEN + 1, stdin);
		tcsetattr(fileno(stdin), TCSANOW, &initial_settings);
		printf("\nYou entered %s\n", password);
	}

	exit(0);
}
```

There are three possibilities for applying changes:

- TCSAFLUSH will change values when current output is complete, but discards any unused available input
- TCSANOW will change values immediately
- TCSADRAIN will change values when current output is complete

Using termios in a C Program

To read one character, including cursor keys:

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

int main(void) {
	struct termios initial_settings, new_settings;
	char key[6] = "x";
	int num_chars;
	tcgetattr(fileno(stdin), &initial_settings);
	new_settings = initial_settings;
	new_settings.c_lflag &= ~ICANON;
	new_settings.c_lflag &= ~ECHO;
	new_settings.c_cc[VMIN] = 1;
	new_settings.c_cc[VTIME] = 0;
	new_settings.c_iflag &= ~ICRNL;

	if (tcsetattr(fileno(stdin), TCSAFLUSH, &new_settings) != 0) {
		fprintf(stderr,"Could not set attributes\n");
		exit(1);
	}

	while (key[0] != 'q') {
		printf("Hit any key: ");
		fflush(stdout);
		num_chars = read(fileno(stdin), key, 3);
		key[num_chars] = '\0';
		if (key[0] == '\015')
			strcpy(key, "Enter");
		else if (strcmp(key, "^[[A") == 0)
			strcpy(key, "Up");
		else if (strcmp(key, "^[[B") == 0)
			strcpy(key, "Down");
		else if (strcmp(key, "^[[D") == 0)
			strcpy(key, "Left");
		else if (strcmp(key, "^[[C") == 0)
			strcpy(key, "Right");

		printf("\nYou entered %s\n", key);
	}

	tcsetattr(fileno(stdin), TCSANOW, &initial_settings);
	exit(0);
}
```

terminfo

terminals are listed by name under lettered directories in */usr/share/terminfo/*

each terminal has its own terminfo file (in termcap everything was combined in one file called */etc/termcap*)

the capabilities are identified by short 2-6 letter tags (2 letters only for termcap)

three types of capabilities

- boolean (e.g. "os" - overstrike is supported)
- numeric (e.g. "cols#80" - number of columns is 80)
- string (e.g. "cup=..." - escape sequence to set cursor position)

terminfo files are binary, compiled using *tic*; use *infocmp* to decompile, show original text entry

toe -a lists all supported terminals in the database

TERM environment variable selects which terminfo entry to use for the shell, editors, etc.

if *clear* command doesn't work then your *TERM* setting is definitely wrong!

Using terminfo in a Shell Script

tput queries and uses terminal capabilities

- tput cols - gives number of columns in current display
- tput lines - gives number of lines in current display
- tput cup 12 53 - moves cursor to row 12 column 53
- tput smso - turns highlighting on
- tput rmso - turns highlighting off

try "man" on "tput"

For example, to use cursor positioning and highlighting:

```
string=
while [ "$string" != 'q' ]
do
      clear
      if [ "$string" != "" ]; then
            tput cup 20 10
            tput smso
            printf "You entered \"$string\""
            tput rmso
      fi
      tput cup 10 10
      printf "Enter something: "
      read string
done
clear
exit 0
```

Another example, using highlighting within a line:

```
bold=$(tput smso)
offbold=$(tput rmso)
printf "${bold}This${offbold} should be highlighted.\n"
printf "So should ${bold}this${offbold}, but not this.\n"
exit 0
```

Using terminfo in a Shell Script

An example to read one character, including cursor keys, using terminfo to obtain the strings for the cursor keys:

```
oldsettings=$(stty -g)
stty -icanon min 1 time 0 -icrnl -echo
enter="$(tput cr)"
up="$(tput kcuu1)"
down="$(tput kcud1)"
right="$(tput kcuf1)"
left="$(tput kcub1)"
tput smkx   # set "keypad send mode", needed for cursor keys
key=
while [ "$key" != "q" ]; do
      printf "Hit any key: "
      key=$(dd bs=3 count=1 2> /dev/null)
      if [ "$key" = "$enter" ]; then
            key="Enter"
      elif [ "$key" = "$up" ]; then
            key="Up"
      elif [ "$key" = "$down" ]; then
            key="Down"
      elif [ "$key" = "$left" ]; then
            key="Left"
      elif [ "$key" = "$right" ]; then
            key="Right"
      fi
      printf "\nYou hit the \"$key\" key\n"
done
stty $oldsettings
```

Using terminfo in a C Program

- initialize by calling setupterm( )
- extract capabilities using tigetflag( ), tigetnum( ), or tigetstr( )
- expand strings with parameters using tparm( )
- output using tputs( ) or putp( )

An example to display the current terminal size (this must be compiled with the "-lncurses" option):

```
#include <stdio.h>
#include <stdlib.h>
#include <term.h>
#include <ncurses.h>

int main(void) {
      int width, height;
      setupterm("vt100", fileno(stdout), (int *)0);
      width = tigetnum("cols");
      height = tigetnum("lines");
      printf("The terminal is %d columns wide and %d lines high\n", width,
      height);
      exit(0);
}
```

An example to move the cursor to row 12, column 53 (this must be compiled with the "-lncurses" option):

```
#include <stdio.h>
#include <stdlib.h>
#include <term.h>
#include <ncurses.h>

int main(void) {
      char *cursor;
      setupterm(NULL, fileno(stdout), (int *)0);
      cursor = tigetstr("cup");
      putp(tparm(cursor, 12, 53));
      exit(0);
}
```

Notice that the first argument to setupterm( ) is NULL; the value of the environment variable TERM will be used

An example to clear the screen and move the cursor to whichever row and column are specified as arguments (this must be compiled with the "-lncurses" option):

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <term.h>
#include <ncurses.h>

int main(int argc, char *argv[]) {
      char *clear, *cursor;
      int row, column;
      sscanf(argv[1], "%d", &row);
      sscanf(argv[2], "%d", &column);
      setupterm(NULL, fileno(stdout), (int *)0);
      clear = tigetstr("clear");
      putp(clear);
      cursor = tigetstr("cup");
      putp(tparm(cursor, row, column));
      exit(0);
}
```

An example to illustrate highlighting (this must be compiled with the "-lncurses" option):

```
#include <stdio.h>
#include <stdlib.h>
#include <term.h>
#include <ncurses.h>

int main(void) {
      char *bold, *offbold;
      setupterm(NULL, fileno(stdout), (int *)0);
      bold = tigetstr("smso");
      offbold = tigetstr("rmso");
      putp(bold);
      printf("This should be highlighted\n");
      putp(offbold);
      exit(0);
}
```

The curses Library

curses is a higher level interface than terminfo/termcap/termios

The higher level routines in the *curses* library provides a terminal independent way to write full-screen applications.

"Half-way" between simple line-based programs and fancy graphical programs.

Main advantage: curses works over almost any type of "serial" connection, e.g.

- regular serial ports
- telnet & rlogin connections
- all types of terminal emulators (just need terminfo entry)
- pseudo-terminals
- real dumb terminals

This makes curses the preferred library for system admin tools that require GUI. (UNIX programmers will go out of their way to avoid GUI.)

curses on Linux

Although using these routines gives us very fine control over the terminal it requires writing a lot of low-level code.

Linux uses **ncurses** (new curses) which is backwards compatible with the original **curses**.

The **curses** interface provides basic input and output methods and multiple windows. **ncurses** adds line and box drawing, color, pads, panels, menus and forms.

To use the **ncurses** library include the following header file:

```
#include <curses.h>
```

and compile and link using:

```
g++ o myapp myapp.cpp lcurses
```

A **window** is a data structure describing a subrectangle of the screen (possibly the entire screen). You can write to and scroll a window independently of other windows.

A **screen** is a subset of windows as large as the terminal screen. The **curscr** structure is an image of what the terminal looks like. Output is performed to the **stdscr** and a call to **refresh()** copies the **stdscr** to the **curscr**.

Canonical manual for curses:
http://csserver.evansville.edu/~amr63/linux/ncurses/ncurses-intro.html

Terminal Input Mode

The input device is in canonical (standard) mode by default. *curses* offers a replacement for much of the termios functionality. The following routines can be used to control the input mode:

```
echo();     // turn on char. echo
noecho();   // turn off char. echo
cbreak();   // raw with signals
nocbreak(); // restore std mode
raw();      // raw w/o signals
noraw();    // restore std mode
```

Curses Input

The following input routines are available:

```
keypad();          // process ESC seq.
getch();           // get a char
getstr(str);       // get a C string
```

getch( ) is intended to be used in **cbreak( )** or **raw( )** mode with or without **noecho( )**.

Curses Color Routines

The following color routines are available:

```
has_colors();           // Does term do color?
start_color();          // init. color support
init_pair(...);         // set up fg/bg pair
attron(COLOR_PAIR(1));  // use as attr.
```

Variable COLORS contains the # of colors and COLOR_PAIRS the # of color pairs.

Multiple Windows

The following window routines are available:

```
newwin();   // set up new win struct
delwin();   // delete window
touchwin(); // flag window as changed
```

There are also routines of the form (**waddch, wmove, wprintw, wrefresh**) that are for working with a specific window.

Basic Curses Routines

All curses programs must start with

```
initscr();
```

which must be called only once. All curses programs must end with

```
endwin();
```

which may be called multiple times. This function exits curses mode. To resume use

```
clearok(stdscr, 1);
refresh();
```

Home (upper left) is (0, 0). Lower right is (LINES-1, COLS-1). Coordinates are in (row, col) format.

To move the cursor:

```
move(y, x);       // to row y, col x
```

To clear the screen:

```
erase();          // blanks to stdscr
clear();          // erase(), redisplay
clrtoeol();       // clear to line end
clrtobot();       // clear to bottom
```

To display output, e.g.:

```
addch(ch);        // display ch
insch(ch);        // insert ch
printw(fmt, var); // format string
insertln();       // insert blank line
beep();           // get user attention
box(win,vch,hch); // draw a box
```

Many also have mv*() forms that perform cursor movement. E.g.,

```
mvaddch(y, x, ch);
```

Each character can have certain attributes (bold, underline, reverse, blink, etc). Attributes are controlled with the attron( ), attroff( ), and attrset( ) routines. See the attron( ) man page for a list of attributes.