

Terminal codes (ANSI/VT100) introduction

Terminal (control) codes are used to issue specific commands to your terminal. This can be related to switching colors or positioning the cursor, i.e. anything that can't be done by the application itself.

How it technically works

A terminal control code is a special sequence of characters that is printed (like any other text). If the terminal understands the code, it won't display the character-sequence, but will perform some action. You can print the codes with a simple `echo` command.

Note: I see codes referenced as "Bash colors" sometimes (several "Bash tutorials" etc...): That's a completely incorrect definition.

The tput command

Because there's a large number of different terminal control languages, usually a system has an intermediate communication layer. The real codes are looked up in a database **for the currently detected terminal type** and you give standardized requests to an [API](#) or (from the shell) to a command.

One of these commands is `tput`. `Tput` accepts a set of acronyms called *capability names* and any parameters, if appropriate, then looks up the correct escape sequences for the detected terminal in the `terminfo` database and prints the correct codes (the terminal hopefully understands).

The codes

In this list I'll focus on ANSI/VT100 control codes for the most common actions - take it as quick reference. The documentation of your terminal or the `terminfo` database is always the preferred source when something is unclear! Also the `tput` acronyms are usually the ones dedicated for ANSI escapes!

I listed only the most relevant codes, of course, any ANSI terminal understands many more! But let's keep the discussion centered on common shell scripting 😊

If I couldn't find a matching ANSI escape, you'll see a 🟡 as the code. Feel free to mail me or fix it.

The ANSI codes always start with the ESC character. ([ASCII](#) 0x1B or octal 033) This isn't part of the list, but **you should avoid using the ANSI codes directly - use the `tput` command!**

All codes that can be used with `tput` can be found in `terminfo(5)`. (on OpenBSD at least) See OpenBSD's `terminfo(5)`

under the [Capabilities](#) section. The *cap-name* is the code to use with `tput`. A description of each code is also provided.

General useful ASCII codes

The **Ctrl-Key** representation is simply associating the non-printable characters from [ASCII](#)

code 1 with the printable (letter) characters from [ASCII](#)

code 65 ("A"). [ASCII](#)

code 1 would be

`^A` (Ctrl-A), while [ASCII](#) code 7 (BEL) would be `^G` (Ctrl-G).

This is a common representation (and input method) and historically comes from one of the VT series of terminals.

Name	decimal	octal	hex	C-escape	Ctrl-Key	Description
BEL	7	007	0x07	\a	^G	Terminal bell

Name	decimal	octal	hex	C-escape	Ctrl-Key	Description
BEL	7	007	0x07	\a	^G	Terminal bell
BS	8	010	0x08	\b	^H	Backspace
HT	9	011	0x09	\t	^I	Horizontal TAB
LF	10	012	0x0A	\n	^J	Linefeed (newline)
VT	11	013	0x0B	\v	^K	Vertical TAB
FF	12	014	0x0C	\f	^L	Formfeed (also: New page NP)
CR	13	015	0x0D	\r	^M	Carriage return
ESC	27	033	0x1B	<none>	^[Escape character
DEL	127	177	0x7F	<none>	<none>	Delete character

Cursor handling

ANSI	terminfo equivalent	Description
[<X> ; <Y> H [<X> ; <Y> f	cup <X> <Y>	Home-positioning to X and Y coordinates ! it seems that ANSI uses 1-1 as home while tput uses 0-0
[H	home	Move cursor to home position (0-0)
7	sc	Save current cursor position
8	rc	Restore saved cursor position
? most likely a normal code like \b	cub1	move left one space (backspace)
VT100 [? 25 l	civis	make cursor invisible
VT100 [? 25 h	cvvis	make cursor visible

Erasing text

ANSI	terminfo equivalent	Description
[K [0 K	e1	Clear line from current cursor position to end of line
[1 K	e11	Clear line from beginning to current cursor position
[2 K	e12 ?	Clear whole line (cursor position unchanged)

General text attributes

ANSI	terminfo equivalent	Description
[0 m	sgr0	Reset all attributes
[1 m	bold	Set "bright" attribute

[0 m	sgr0	Reset all attributes
[1 m	bold	Set "bright" attribute
[2 m	dim	Set "dim" attribute
[3 m	smso	Set "standout" attribute
[4 m	set smul unset rmul ?	Set "underscore" (underlined text) attribute
[5 m	blink	Set "blink" attribute
[7 m	rev	Set "reverse" attribute
[8 m	invis	Set "hidden" attribute

Foreground coloring

ANSI	terminfo equivalent	Description
[3 0 m	setaf 0	Set foreground to color #0 - black
[3 1 m	setaf 1	Set foreground to color #1 - red
[3 2 m	setaf 2	Set foreground to color #2 - green
[3 3 m	setaf 3	Set foreground to color #3 - yellow
[3 4 m	setaf 4	Set foreground to color #4 - blue
[3 5 m	setaf 5	Set foreground to color #5 - magenta
[3 6 m	setaf 6	Set foreground to color #6 - cyan
[3 7 m	setaf 7	Set foreground to color #7 - white
[3 9 m	setaf 9	Set default color as foreground color

Background coloring

ANSI	terminfo equivalent	Description
[4 0 m	setab 0	Set background to color #0 - black
[4 1 m	setab 1	Set background to color #1 - red
[4 2 m	setab 2	Set background to color #2 - green
[4 3 m	setab 3	Set background to color #3 - yellow
[4 4 m	setab 4	Set background to color #4 - blue
[4 5 m	setab 5	Set background to color #5 - magenta
[4 6 m	setab 6	Set background to color #6 - cyan
[4 7 m	setab 7	Set background to color #7 - white
[4 9 m	setaf 9	Set default color as background color

Misc codes

Misc codes

Save/restore screen

Used capabilities: `smcup` , `rmcup`

You've undoubtedly already encountered programs that restore the terminal contents after they do their work (like `vim`). This can be done by the following commands:

```
# save, clear screen
tput smcup
clear

# example "application" follows...
read -n1 -p "Press any key to continue..."
# example "application" ends here

# restore
tput rmcup
```

These features require that certain capabilities exist in your `termcap/terminfo`. While `xterm` and most of its clones (`rxvt` , `urxvt` , etc) will support the instructions, your operating system may not include references to them in its default `xterm` profile. (FreeBSD, in particular, falls into this category.) If ``tput smcup`` appears to do nothing for you, and you don't want to modify your system `termcap/terminfo` data, and you KNOW that you are using a compatible `xterm` application, the following may work for you:

```
echo -e '\033[?47h' # save screen
echo -e '\033[?47l' # restore screen
```

Certain software uses these codes (via their `termcap` capabilities) as well. You may have seen the screen save/restore in `less` , `vim` , `top` , `screen` and others. Some of these applications may also provide configuration options to `*disable*` this behaviour. For example, `less` has a `-X` option for this, which can also be set in an environment variable:

```
export LESS=X
less /path/to/file
```

Similarly, `vim` can be configured not to "restore" the screen by adding the following to your `~/.vimrc` :

```
set t_ti= t_te=
```

Additional colors

Some terminal emulators support many additional colors. The most common extension used by `xterm`-compatible terminals supports 256 colors. These can be generated by `tput` with `seta{f,b} [0-255]` when the `TERM` value has a `-256color` suffix. Konsole supports full 24-bit colors, and as of KDE 4.9 any X11 color code can be written directly into a special escape sequence. Other terminals may offer similar extensions. Few, if any programs are able to make use of anything beyond 256, and `tput` doesn't know about them. Colors beyond 16 usually only apply to modern terminal emulators running in graphical environments.

The Virtual Terminal implemented in the Linux kernel supports only 16 colors, and the usual default `terminfo` entry for `TERM=linux` defines only 8. There is sometimes an alternate "linux-16color" that you can switch to, to get the other 8 colors.

Bash examples

Hardcoded colors

Bash examples

Hardcoded colors

```
printf '%b\n' 'It is \033[31mnot\033[39m intelligent to use \033[32mhardcoded ANSI\033[39m code'
```

Colors using tput

Directly inside the echo:

```
echo "TPUT is a $(tput setaf 2)nice$(tput setaf 9) and $(tput setaf 5)user friendly$(tput setaf
```

With preset variables:

```
COL_NORM="$(tput setaf 9)"
COL_RED="$(tput setaf 1)"
COL_GREEN="$(tput setaf 2)"
echo "It's ${COL_RED}red${COL_NORM} and ${COL_GREEN}green${COL_NORM} - have you seen?"
```

Misc

HOME function

```
home() {
  # yes, actually not much shorter ;-)
  tput home
}
```

Silly but nice effect

```
#!/bin/bash

DATA[0]="  _/  _/  _/  _/  _/  _/"
DATA[1]=" _/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/"
DATA[2]="  _/  _/  _/  _/  _/  _/  _/  _/  _/  _/"
DATA[3]=" _/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/"
DATA[4]="  _/  _/  _/_/_/_/_/_/_/_/_/_/_/_/_/"

# virtual coordinate system is X*Y ${#DATA} * 5

REAL_OFFSET_X=0
REAL_OFFSET_Y=0

draw_char() {
  V_COORD_X=$1
  V_COORD_Y=$2

  tput cup $((REAL_OFFSET_Y + V_COORD_Y)) $((REAL_OFFSET_X + V_COORD_X))

  printf %c ${DATA[V_COORD_Y]:V_COORD_X:1}
}

trap 'exit 1' INT TERM
```

```

trap 'exit 1' INT TERM
trap 'tput setaf 9; tput cvvis; clear' EXIT

tput civis
clear

while ;; do

for ((c=1; c <= 7; c++)); do
  tput setaf $c
  for ((x=0; x<${#DATA[0]}; x++)); do
    for ((y=0; y<=4; y++)); do
      draw_char $x $y
    done
  done
done
done
done

```

Mandelbrot set

This is a slightly modified version of Charles Cooke's colorful Mandelbrot plot scripts ([original w/ screenshot](#)) – ungolfed, optimized a bit, and without hard-coded terminal escapes. The `colorBox` function is memoized to collect `tput` output only when required and output a new escape only when a color change is needed. This limits the number of `tput` calls to at most 16, and reduces raw output by more than half. The `doBash` function uses integer arithmetic, but is still ksh93-compatible (run as e.g. `bash ./mandelbrot` to use it). The ksh93-only floating-point `doKsh` is almost 10x faster than `doBash` (thus the ksh shebang by default), but uses only features that don't make the Bash parser crash.

```

#!/usr/bin/env ksh

# Charles Cooke's 16-color Mandelbrot
# http://earth.gkhs.net/ccooke/shell.html
# Combined Bash/ksh93 flavors by Dan Douglas (ormaaaj)

function doBash {
  typeset P Q X Y a b c i v x y
  for ((P=10**8,Q=P/100,X=320*Q/cols,Y=210*Q/lines,y=-105*Q,v=-220*Q,x=v;y<105*Q;x=v,y+=Y
    for ((;x<P;a=b=i=c=0,x+=X)); do
      for ((;a**2+b**2<4*P**2&&i++<99;a=((c=a)**2-b**2)/P+x,b=2*c*b/P+y)); do
        done
        colorBox $((i<99?i%16:0))
      done
    done
  echo
done
}

function doKsh {
  integer i
  float a b c x=2.2 y=-1.05 X=3.2/cols Y=2.1/lines
  while
    for ((a=b=i=0;(c=a)**2+b**2<=2&&i++<99&&(a=a**2-b**2+x,b=2*c*b+y)); do :
      done
    . colorBox $((i<99?i%16:0))

```

```

        for ((a=b=i=0;(c=a)**2+b**2<=2&&i++<99&&(a=a**2-b**2+x,b=2*c*b+y)); do :
        done
        . colorBox $((i<99?i%16:0))
        if ((x<1?(x+=X):(y+=Y,x=-2.2)); then
                print
                ((y<1.05))
        fi
        do :
done
}

function colorBox {
    (($1==lastclr)) || printf %s "${colrs[lastclr=$1]}:=$(tput setaf "$1")}"
    printf '\u2588'
}

unset -v lastclr
((cols=$(tput cols)-1, lines=$(tput lines)))
typeset -a colrs
trap 'tput sgr0; echo' EXIT
${KSH_VERSION+. doKsh} ${BASH_VERSION+doBash}

```

A much more sophisticated version by Roland Mainz can be found [here](#)

Discussion

Lucas H, [2011/07/28 06:37](#)

RE: your "Note: I found no code to entirely erase the current line ("delete line" is something else!). It might be a combination of positioning the cursor and erase to the end of line."

Try this: [2 K e]2 Clear whole line

James, [2011/09/01 00:12](#)

In the table showing

[3 9 m setaf 9 Set default foreground color

the Description "Set default foreground color" is ambiguous.

That phrase could mean either that the commands will 1) store the value of a specified color as the "default" color value, or that 2) a stored "default" color value will be used to re-set the current foreground or background color to a new value. Which is it? In one case there can be a visible change on the screen. In the other case, the will never be a visible change on the screen.

As it is, some people will create termcap files which gratuitously reset the display to the "default" colors, which makes using custom foreground and background colors impossible. Of course, this is just mean, and requires rewriting the termcap file.

Also, the Descriptions of the "Dim", "Bright", and "Reverse" attributes could actually say what these are suppose to do. For instance, what is suppose to happen when setting both "Dim" and "Bright"? Or, does "Reverse" apply to both the foreground and background colors? Does "Reverse" mean to exchange the foreground and background

Also, the Descriptions of the Dim , Bright , and Reverse attributes could actually say what these are suppose to do. For instance, what is suppose to happen when setting both "Dim" and "Bright"? Or, does "Reverse" apply to both the foreground and background colors? Does "Reverse" mean to exchange the foreground and background colors? Or to set some kind of "complement" color to each of the foreground and background?

These "Descriptions" that do not describe are not useful.

James

Constantine, 2011/09/21 14:43

print shortcuts for all ansi codes, NB: please add plus plus in for statements!

```
ansi-test()
{
  for a in 0 1 4 5 7; do
    echo "a=$a "
    for (( f=0; f<=9; f++ )) ; do
      for (( b=0; b<=9; b++ )) ; do
        #echo -ne "f=$f b=$b"
        echo -ne "\\033[${a};3${f};4${b}m"
        echo -ne "\\033[${a};3${f};4${b}m"
        echo -ne "\\033[0m "
      done
    done
    echo
    done
  done
  echo
}
```

Aubrey Bourke , 2011/12/19 02:38

Hi,

Very cool tutorial. I recently purchased a beagleboard XM, so this site is a perfect place to start serial port programming.

And the "silly but nice effects" is awesome. I love it!

Here's a link to a "cool splash screen for my website". Its just a Java animation... (open with Java web start - jws)

file:///home/aubrey/Downloads/circlescroller.jnlp

Best Regards.

Jan Schampera, 2011/12/21 12:35

Hi,

thank you 😊

I don't think this link will work for anybody except you (file:) 😞

I don't think this link will work for anybody except you (me.) 😊

Bill Gradwohl, [2012/04/08 01:57](#)

This describes things from the display end. What about the keyboard? How does someone read the codes from the keyboard and figure out that the user pressed the up arrow key, for example?

I'm interested in this for using the bash `read -s -n 1` mechanism to bring in keystrokes 1 character at a time and then try to figure out what key the user pressed. Up arrow for example is `\E[A`. I want to get the entire list of possible character combinations that are legitimate for a given environment.

The `infocmp` utility can dump the terminfo for a particular entity (`xterm`, `linux`, etc) but I can't find the equivalent for a keyboard.

Jan Schampera, [2012/04/21 12:45](#)

A very good question. Sorry, I can't answer it. I think there are no such things as "standardized" key codes.

Ruthard Baudach, [2013/04/03 21:35](#)

well, just use `read`!

`read` does not only read the input from the keyboard, but reflects it on the terminal – resulting in the keycodes you are looking for.

I used my findings for the following python script: (sorry for not using bash)

```
#!/usr/bin/env python

import sys,termios, time

## Font attributes ##
# off
off = '\x1b[0m' # off
default = '\x1b[39m' # default foreground
DEFAULT = '\x1b[49m' # default background
#
bd = '\x1b[1m' # bold
ft = '\x1b[2m' # faint
st = '\x1b[3m' # standout
ul = '\x1b[4m' # underlined
bk = '\x1b[5m' # blink
rv = '\x1b[7m' # reverse
hd = '\x1b[8m' # hidden
nost = '\x1b[23m' # no standout
noul = '\x1b[24m' # no underlined
nobk = '\x1b[25m' # no blink
norv = '\x1b[27m' # no reverse
# colors
black = '\x1b[30m'
BLACK = '\x1b[40m'
red = '\x1b[31m'
RED = '\x1b[41m'
green = '\x1b[32m'
```

```
red = '\x1b[31m'
RED = '\x1b[41m'
green = '\x1b[32m'
GREEN = '\x1b[42m'
yellow = '\x1b[33m'
YELLOW = '\x1b[43m'
blue = '\x1b[34m'
BLUE = '\x1b[44m'
magenta = '\x1b[35m'
MAGENTA = '\x1b[45m'
cyan = '\x1b[36m'
CYAN = '\x1b[46m'
white = '\x1b[37m'
WHITE = '\x1b[47m'
# light colors
dgray = '\x1b[90m'
DGRAY = '\x1b[100m'
lred = '\x1b[91m'
LRED = '\x1b[101m'
lgreen = '\x1b[92m'
LGREEN = '\x1b[102m'
lyellow = '\x1b[93m'
LYELLOW = '\x1b[103m'
lblue = '\x1b[94m'
LBLUE = '\x1b[104m'
lmagenta = '\x1b[95m'
LMAGENTA = '\x1b[105m'
lcyan = '\x1b[96m'
LCYAN = '\x1b[106m'
lgray = '\x1b[97m'
LGRAY = '\x1b[107m'

## 256 colors ##
# \x1b[38;5;#m foreground, # = 0 - 255
# \x1b[48;5;#m background, # = 0 - 255
## True Color ##
# \x1b[38;2;r;g;b;#m r = red, g = green, b = blue foreground
# \x1b[48;2;r;g;b;#m r = red, g = green, b = blue background

# -----
# prepare terminal settings
fd = sys.stdin.fileno()
old_settings = termios.tcgetattr(fd)
new_settings = termios.tcgetattr(fd)
new_settings[3] &= ~termios.ICANON
new_settings[3] &= ~termios.ECHO

# -----
def clear(what='screen'):
    """
    erase functions:
        what: screen => erase screen and go home
              line  => erase line and go to start of line
              bos   => erase to begin of screen
              eos   => erase to end of screen
              bol   => erase to begin of line
              eol   => erase to end of line
```

```
        eos    => erase to end of screen
        bol    => erase to begin of line
        eol    => erase to end of line
    ...
clear = {
    'screen': '\x1b[2J\x1b[H',
    'line': '\x1b[2K\x1b[G',
    'bos': '\x1b[1J',
    'eos': '\x1b[J',
    'bol': '\x1b[1K',
    'eol': '\x1b[K',
}
sys.stdout.write(clear[what])
sys.stdout.flush()

# -----
def move(pos):
    ...

    move cursor to pos
    pos = tuple (x,y)
    ...

    x,y = pos
    sys.stdout.write('\x1b[{};{}H'.format(str(x),str(y)))
    sys.stdout.flush()

# -----
def put(*args):
    ...

    put text on on screen
    a tuple as first argument tells absolute position for the text
    does not change cursor position
    args = list of optional position, formatting tokens and strings
    ...

    args = list(args)
    if type(args[0]) == type(()):
        x,y = args[0]
        del args[0]
        args.insert(0, '\x1b[{};{}H'.format(str(x),str(y)))
    args.insert(0, '\x1b[s')
    args.append('\x1b[u')
    sys.stdout.write(''.join(args))
    sys.stdout.flush()

# -----
def write(*args):
    ...

    writes text on on screen
    a tuple as first argument gives the relative position to current cursor position
    does change cursor position
    args = list of optional position, formatting tokens and strings
    ...

    args = list(args)
    if type(args[0]) == type(()):
        pos = []
        x,y = args[0]
        if x > 0:
            pos.append('\x1b[{}A'.format(str(x)))
```

```
x,y = args[0]
if x > 0:
    pos.append('\x1b[{}A'.format(str(x)))
elif x < 0:
    pos.append('\x1b[{}B'.format(abs(str(x))))
if y > 0:
    pos.append('\x1b[{}C'.format(str(y)))
elif y < 0:
    pos.append('\x1b[{}D'.format(abs(str(y))))
del args[0]
args = pos + args
sys.stdout.write(''.join(args))
sys.stdout.flush()

# -----
def getch():
    '''
    Get character.
    '''
    # get character
    try:
        termios.tcsetattr(fd,termios.TCSANOW,new_settings)
        ch = sys.stdin.read(1)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
    # return
    return ch

# -----

if __name__ == '__main__':
    clear()
    esc_mode = False
    esc_string = ''
    esc_codes = {
        'A': 'Up',
        'B': 'Down',
        'C': 'Right',
        'D': 'Left',
        'F': 'End',
        'H': 'Pos1',
        '2~': 'Ins',
        '3~': 'Del',
        '5~': 'PgUp',
        '6~': 'PdDown',
        'OP': 'F1',
        'OQ': 'F2',
        'OR': 'F3',
        'OS': 'F4',
        '[15~': 'F5',
        '[17~': 'F6',
        '[18~': 'F7',
        '[19~': 'F8',
        '[20~': 'F9',
        '[21~': 'F10',
```

```
'[19~': 'F8',
'[20~': 'F9',
'[21~': 'F10',
'[23~': 'F11',
'[24~': 'F12',
'[29~': 'Apps',
'[34~': 'Win',
'[1;2A': 'S-Up',
'[1;2B': 'S-Down',
'[1;2C': 'S-Right',
'[1;2D': 'S-Left',
'[1;2F': 'S-End',
'[1;2H': 'S-Pos1',
'[2;2~': 'S-Ins',
'[3;2~': 'S-De1',
'[5;2~': 'S-PgUp',
'[6;2~': 'S-PdDown',
'[1;2P': 'S-F1',
'[1;2Q': 'S-F2',
'[1;2R': 'S-F3',
'[1;2S': 'S-F4',
'[15;2~': 'S-F5',
'[17;2~': 'S-F6',
'[18;2~': 'S-F7',
'[19;2~': 'S-F8',
'[20;2~': 'S-F9',
'[21;2~': 'S-F10',
'[23;2~': 'S-F11',
'[24;2~': 'S-F12',
'[29;2~': 'S-Apps',
'[34;2~': 'S-Win',
'[1;3A': 'M-Up',
          '[1;3B': 'M-Down',
'[1;3C': 'M-Right',
'[1;3D': 'M-Left',
'[1;3F': 'M-End',
'[1;3H': 'M-Pos1',
'[2;3~': 'M-Ins',
'[3;3~': 'M-De1',
'[5;3~': 'M-PgUp',
'[6;3~': 'M-PdDown',
'[1;3P': 'M-F1',
'[1;3Q': 'M-F2',
'[1;3R': 'M-F3',
'[1;3S': 'M-F4',
'[15;3~': 'M-F5',
'[17;3~': 'M-F6',
'[18;3~': 'M-F7',
'[19;3~': 'M-F8',
'[20;3~': 'M-F9',
'[21;3~': 'M-F10',
'[23;3~': 'M-F11',
'[24;3~': 'M-F12',
'[29;3~': 'M-Apps',
'[34;3~': 'M-Win',
'[1;5A': 'C-Up',
'[1;5B': 'C-Down',
```

```
[34;5~ : M-Win ,
'[1;5A': 'C-Up',
'[1;5B': 'C-Down',
'[1;5C': 'C-Right',
'[1;5D': 'C-Left',
'[1;5F': 'C-End',
'[1;5H': 'C-Pos1',
'[2;5~': 'C-Ins',
'[3;5~': 'C-Del',
'[5;5~': 'C-PgUp',
'[6;5~': 'C-PdDown',
'[1;5P': 'C-F1',
'[1;5Q': 'C-F2',
'[1;5R': 'C-F3',
'[1;5S': 'C-F4',
'[15;5~': 'C-F5',
'[17;5~': 'C-F6',
'[18;5~': 'C-F7',
'[19;5~': 'C-F8',
'[20;5~': 'C-F9',
'[21;5~': 'C-F10',
'[23;5~': 'C-F11',
'[24;5~': 'C-F12',
'[29;5~': 'C-Apps',
'[34;5~': 'C-Win',
'[1;6A': 'S-C-Up',
'[1;6B': 'S-C-Down',
'[1;6C': 'S-C-Right',
'[1;6D': 'S-C-Left',
'[1;6F': 'S-C-End',
'[1;6H': 'S-C-Pos1',
'[2;6~': 'S-C-Ins',
      '[3;6~': 'S-C-Del',
'[5;6~': 'S-C-PgUp',
'[6;6~': 'S-C-PdDown',
'[1;6P': 'S-C-F1',
'[1;6Q': 'S-C-F2',
'[1;6R': 'S-C-F3',
'[1;6S': 'S-C-F4',
'[15;6~': 'S-C-F5',
'[17;6~': 'S-C-F6',
'[18;6~': 'S-C-F7',
'[19;6~': 'S-C-F8',
'[20;6~': 'S-C-F9',
'[21;6~': 'S-C-F10',
'[23;6~': 'S-C-F11',
'[24;6~': 'S-C-F12',
'[29;6~': 'S-C-Apps',
'[34;6~': 'S-C-Win',
'[1;7A': 'C-M-Up',
'[1;7B': 'C-M-Down',
'[1;7C': 'C-M-Right',
'[1;7D': 'C-M-Left',
'[1;7F': 'C-M-End',
'[1;7H': 'C-M-Pos1',
'[2;7~': 'C-M-Ins',
'[3;7~': 'C-M-Del',
```

```
[4;7~: 'C-M-F5',  
'[2;7~: 'C-M-Ins',  
'[3;7~: 'C-M-Del',  
'[5;7~: 'C-M-PgUp',  
'[6;7~: 'C-M-PdDown',  
'[1;7P: 'C-M-F1',  
'[1;7Q: 'C-M-F2',  
'[1;7R: 'C-M-F3',  
'[1;7S: 'C-M-F4',  
'[15;7~: 'C-M-F5',  
'[17;7~: 'C-M-F6',  
'[18;7~: 'C-M-F7',  
'[19;7~: 'C-M-F8',  
'[20;7~: 'C-M-F9',  
'[21;7~: 'C-M-F10',  
'[23;7~: 'C-M-F11',  
'[24;7~: 'C-M-F12',  
'[29;7~: 'C-M-Apps',  
'[34;7~: 'C-M-Win',  
}  
# 8 wäre S-C-M
```

```
ctrl_codes = {  
    0: 'C-2',  
    1: 'C-A',  
    2: 'C-B',  
    3: 'C-C',  
    4: 'C-D',  
    5: 'C-E',  
    6: 'C-F',  
    7: 'C-G',  
    8: 'C-H',  
    9: 'C-I',  
    10: 'C-J',  
    11: 'C-K',  
    12: 'C-L',  
    13: 'C-M',  
    14: 'C-N',  
    15: 'C-O',  
    16: 'C-P',  
    17: 'C-Q',  
    18: 'C-R',  
    19: 'C-S',  
    20: 'C-T',  
    21: 'C-U',  
    22: 'C-V',  
    23: 'C-W',  
    24: 'C-X',  
    25: 'C-Y',  
    26: 'C-Z',  
    27: 'C-3',  
    29: 'C-5',  
    30: 'C-6',  
    31: 'C-7',  
}
```

```
while True:
```

```

while True:
    move((1,1))
    clear('line')
    put((1,1),green,':',off)
    move((1,3))
    ch = getch()
    if esc_mode:
        esc_string += ch
        # esc string terminators
        if ch in ['A','B','C','D','F','H','P','Q','R','S','~']:
            esc_mode = False
            move((2,0))
            clear('line')
            put((2,5),esc_codes[esc_string])
            esc_string = ''
        elif ch == '\x1b':
            esc_mode = False
    else:
        # esc mode
        if ch == '\x1b':
            esc_mode = True
        # ctrl
        elif ord(ch) in ctrl_codes.keys():
            move((2,0))
            clear('line')
            put((2,5),ctrl_codes[ord(ch)])

    move((2,0))
    put((2,3),str(ch))

```

Iskren Hadzhinedev

, [2013/11/04 13:33](#)

If you're using X, you can get keycodes from the keyboard with the 'xev' program; it opens a window that prints in the terminal every event (mouse move, mouse button press, keypress, keyrelease, etc). I know I'm more than a year late, but google brought me here, so hopefully someone will find this useful. Cheers.

JK Benedict, [2014/08/30 05:36](#)

First - thank you for this article as I have written a sub-routine for various *nix and non-nix systems to parse ANSI (as best as possible). Point is - I WORSHIP THIS OVERVIEW - especially when I come across individuals interested in making the most of bash, etc.

Second - I get to contribute!

@Bill – From bash, leverage the read command. I've included a few links for reference, but the general idea is that it can be used for "Hey, type in a something and press enter" to being nested in a loop condition to "trap" (that is a term you will want to look at) single key strokes. The command even goes as far to give a "timeout" if the user doesn't press any key!

http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_08_02.html

http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_08_02.html

<http://www.unix.com/shell-programming-and-scripting/140231-bash-keypress-read-single-character.html>

and this sorta brings the previous links together in a practical example:

<http://top-scripts.blogspot.com/2011/01/blog-post.html>

HTH! -jkbs @xenfomation

Albert25, 2015/07/10 10:46

Quickly see the foreground/background colors:

```
for b in {0..7} 9; do for f in {0..7} 9; do for attr in "" bold; do echo -e "$(tput setab $b; tput setaf $f; [ -n "$attr" ] && tput $attr) $f ON $b" && done
```

Or the same on several lines for readability:

```
for b in {0..7} 9; do
  for f in {0..7} 9; do
    for attr in "" bold; do
      echo -e "$(tput setab $b; tput setaf $f; [ -n "$attr" ] && tput $attr) $f ON $b" &&
    done
  done
done
```