NAME

      Term::ANSIColor - Color screen output using ANSI escape sequences

SYNOPSIS

```
use Term::ANSIColor;
print color('bold blue');
print "This text is bold blue.\n";
print color('reset');
print "This text is normal.\n";
print colored("Yellow on magenta.", 'yellow on_magenta'), "\n";
print "This text is normal.\n";
print colored(['yellow on_magenta'], 'Yellow on magenta.', "\n");
print colored(['red on_bright_yellow'], 'Red on bright yellow.', "\n");
print colored(['bright_red on_black'], 'Bright red on black.', "\n");
print "\n";

# Map escape sequences back to color names.
use Term::ANSIColor 1.04 qw(uncolor);
my $names = uncolor('01;31');
print join(q{ }, @{$names}), "\n";

# Strip all color escape sequences.
use Term::ANSIColor 2.01 qw(colorstrip);
print colorstrip("\e[1mThis is bold\e[0m"), "\n";

# Determine whether a color is valid.
use Term::ANSIColor 2.02 qw(colorvalid);
my $valid = colorvalid('blue bold', 'on_magenta');
print "Color string is ", $valid ? "valid\n" : "invalid\n";

# Create new aliases for colors.
use Term::ANSIColor 4.00 qw(coloralias);
coloralias('alert', 'red');
print "Alert is ", coloralias('alert'), "\n";
print colored("This is in red.", 'alert'), "\n";

use Term::ANSIColor qw(:constants);
print BOLD, BLUE, "This text is in bold blue.\n", RESET;

use Term::ANSIColor qw(:constants);
{
local $Term::ANSIColor::AUTORESET = 1;
print BOLD BLUE "This text is in bold blue.\n";
print "This text is normal.\n";
}

use Term::ANSIColor 2.00 qw(:pushpop);
print PUSHCOLOR RED ON_GREEN "This text is red on green.\n";
print PUSHCOLOR BRIGHT_BLUE "This text is bright blue on green.\n";
print RESET BRIGHT_BLUE "This text is just bright blue.\n";
print POPCOLOR "Back to red on green.\n";
print LOCALCOLOR GREEN ON_BLUE "This text is green on blue.\n";
print "This text is red on green.\n";
{
local $Term::ANSIColor::AUTOLOCAL = 1;
```

```
    print ON_BLUE "This text is red on blue.\n";
    print "This text is red on green.\n";
    }
    print POPCOLOR "Back to whatever we started as.\n";
```

DESCRIPTION

This module has two interfaces, one through *color()* and *colored()* and the other through constants. It also offers the utility functions *uncolor()*, *colorstrip()*, *colorvalid()*, and *coloralias()*, which have to be explicitly imported to be used (see ''SYNOPSIS'').

See ''COMPATIBILITY'' for the versions of Term::ANSIColor that introduced particular features and the versions of Perl that included them.

Supported Colors

Terminal emulators that support color divide into three types: ones that support only eight colors, ones that support sixteen, and ones that support 256. This module provides the ANSI escape codes for all of them. These colors are referred to as ANSI colors 0 through 7 (normal), 8 through 15 (16-color), and 16 through 255 (256-color).

Unfortunately, interpretation of colors 0 through 7 often depends on whether the emulator supports eight colors or sixteen colors. Emulators that only support eight colors (such as the Linux console) will display colors 0 through 7 with normal brightness and ignore colors 8 through 15, treating them the same as white. Emulators that support 16 colors, such as gnome-terminal, normally display colors 0 through 7 as dim or darker versions and colors 8 through 15 as normal brightness. On such emulators, the ''normal'' white (color 7) usually is shown as pale grey, requiring bright white (15) to be used to get a real white color. Bright black usually is a dark grey color, although some terminals display it as pure black. Some sixteen-color terminal emulators also treat normal yellow (color 3) as orange or brown, and bright yellow (color 11) as yellow.

Following the normal convention of sixteen-color emulators, this module provides a pair of attributes for each color. For every normal color (0 through 7), the corresponding bright color (8 through 15) is obtained by prepending the string `bright_` to the normal color name. For example, `red` is color 1 and `bright_red` is color 9. The same applies for background colors: `on_red` is the normal color and `on_bright_red` is the bright color. Capitalize these strings for the constant interface.

For 256-color emulators, this module additionally provides `ansi0` through `ansi15`, which are the same as colors 0 through 15 in sixteen-color emulators but use the 256-color escape syntax, `grey0` through `grey23` ranging from nearly black to nearly white, and a set of RGB colors. The RGB colors are of the form `rgbRGB` where *R*, *G*, and *B* are numbers from 0 to 5 giving the intensity of red, green, and blue. `on_` variants of all of these colors are also provided. These colors may be ignored completely on non-256-color terminals or may be misinterpreted and produce random behavior. Additional attributes such as blink, italic, or bold may not work with the 256-color palette.

There is unfortunately no way to know whether the current emulator supports more than eight colors, which makes the choice of colors difficult. The most conservative choice is to use only the regular colors, which are at least displayed on all emulators. However, they will appear dark in sixteen-color terminal emulators, including most common emulators in UNIX X environments. If you know the display is one of those emulators, you may wish to use the bright variants instead. Even better, offer the user a way to configure the colors for a given application to fit their terminal emulator.

Function Interface

The function interface uses attribute strings to describe the colors and text attributes to assign to text. The recognized non-color attributes are clear, reset, bold, dark, faint, italic, underline, underscore, blink, reverse, and concealed. Clear and reset (reset to default attributes), dark and faint (dim and saturated), and underline and underscore are equivalent, so use whichever is the most intuitive to you.

Note that not all attributes are supported by all terminal types, and some terminals may not support any of these sequences. Dark and faint, italic, blink, and concealed in particular are frequently not implemented.

The recognized normal foreground color attributes (colors 0 to 7) are:

```
black red green yellow blue magenta cyan white
```

The corresponding bright foreground color attributes (colors 8 to 15) are:

```
bright_black bright_red bright_green bright_yellow
bright_blue bright_magenta bright_cyan bright_white
```

The recognized normal background color attributes (colors 0 to 7) are:

```
on_black on_red on_green on yellow
on_blue on_magenta on_cyan on_white
```

The recognized bright background color attributes (colors 8 to 15) are:

```
on_bright_black on_bright_red on_bright_green on_bright_yellow
on_bright_blue on_bright_magenta on_bright_cyan on_bright_white
```

For 256-color terminals, the recognized foreground colors are:

```
ansi0 .. ansi15
grey0 .. grey23
```

plus `rgb`*RGB* for *R*, *G*, and *B* values from 0 to 5, such as `rgb000` or `rgb515`. Similarly, the recognized background colors are:

```
on_ansi0 .. on_ansi15
on_grey0 .. on_grey23
```

plus `on_rgb`*RGB* for *R*, *G*, and *B* values from 0 to 5.

For any of the above listed attributes, case is not significant.

Attributes, once set, last until they are unset (by printing the attribute `clear` or `reset`). Be careful to do this, or otherwise your attribute will last after your script is done running, and people get very annoyed at having their prompt and typing changed to weird colors.

color(ATTR[, ATTR ...])
> *color()* takes any number of strings as arguments and considers them to be space-separated lists of attributes. It then forms and returns the escape sequence to set those attributes. It doesn't print it out, just returns it, so you'll have to print it yourself if you want to. This is so that you can save it as a string, pass it to something else, send it to a file handle, or do anything else with it that you might care to. *color()* throws an exception if given an invalid attribute.

colored(STRING, ATTR[, ATTR ...])
colored(ATTR-REF, STRING[, STRING...])
> As an aid in resetting colors, *colored()* takes a scalar as the first argument and any number of attribute strings as the second argument and returns the scalar wrapped in escape codes so that the attributes will be set as requested before the string and reset to normal after the string. Alternately, you can pass a reference to an array as the first argument, and then the contents of that array will be taken as attributes and color codes and the remainder of the arguments as text to colorize.
>
> Normally, *colored()* just puts attribute codes at the beginning and end of the string, but if you set `$Term::ANSIColor::EACHLINE` to some string, that string will be considered the line delimiter and the attribute will be set at the beginning of each line of the passed string and reset at the end of each line. This is often desirable if the output contains newlines and you're using background colors, since a background color that persists across a newline is often interpreted by the terminal as providing the default background color for the next line. Programs like pagers can also be confused by attributes that span lines. Normally you'll want to set `$Term::ANSIColor::EACHLINE` to `"\n"` to use this feature.

uncolor(ESCAPE)
> *uncolor()* performs the opposite translation as *color()*, turning escape sequences into a list of strings corresponding to the attributes being set by those sequences.

colorstrip(STRING[, STRING ...])
>   *colorstrip()* removes all color escape sequences from the provided strings, returning the modified strings separately in array context or joined together in scalar context. Its arguments are not modified.

colorvalid(ATTR[, ATTR ...])
>   *colorvalid()* takes attribute strings the same as *color()* and returns true if all attributes are known and false otherwise.

coloralias(ALIAS[, ATTR])
>   If ATTR is specified, *coloralias()* sets up an alias of ALIAS for the standard color ATTR. From that point forward, ALIAS can be passed into *color()*, *colored()*, and *colorvalid()* and will have the same meaning as ATTR. One possible use of this facility is to give more meaningful names to the 256-color RGB colors. Only alphanumerics, ., _, and – are allowed in alias names.
>
>   If ATTR is not specified, *coloralias()* returns the standard color name to which ALIAS is aliased, if any, or undef if ALIAS does not exist.
>
>   This is the same facility used by the ANSI_COLORS_ALIASES environment variable (see "ENVIRONMENT" below) but can be used at runtime, not just when the module is loaded.
>
>   Later invocations of *coloralias()* with the same ALIAS will override earlier aliases. There is no way to remove an alias.
>
>   Aliases have no effect on the return value of *uncolor()*.
>
>   **WARNING**: Aliases are global and affect all callers in the same process. There is no way to set an alias limited to a particular block of code or a particular object.

Constant Interface

Alternately, if you import `:constants`, you can use the following constants directly:

```
CLEAR RESET BOLD DARK
FAINT ITALIC UNDERLINE UNDERSCORE
BLINK REVERSE CONCEALED

BLACK RED GREEN YELLOW
BLUE MAGENTA CYAN WHITE
BRIGHT_BLACK BRIGHT_RED BRIGHT_GREEN BRIGHT_YELLOW
BRIGHT_BLUE BRIGHT_MAGENTA BRIGHT_CYAN BRIGHT_WHITE

ON_BLACK ON_RED ON_GREEN ON_YELLOW
ON_BLUE ON_MAGENTA ON_CYAN ON_WHITE
ON_BRIGHT_BLACK ON_BRIGHT_RED ON_BRIGHT_GREEN ON_BRIGHT_YELLOW
ON_BRIGHT_BLUE ON_BRIGHT_MAGENTA ON_BRIGHT_CYAN ON_BRIGHT_WHITE
```

These are the same as color('attribute') and can be used if you prefer typing:

```
print BOLD BLUE ON_WHITE "Text", RESET, "\n";
```

to

```
print colored ("Text", 'bold blue on_white'), "\n";
```

(Note that the newline is kept separate to avoid confusing the terminal as described above since a background color is being used.)

If you import `:constants256`, you can use the following constants directly:

```
ANSI0 .. ANSI15
GREY0 .. GREY23

RGBXYZ (for X, Y, and Z values from 0 to 5, like RGB000 or RGB515)

ON_ANSI0 .. ON_ANSI15
```

```
ON_GREY0 .. ON_GREY23

ON_RGBXYZ (for X, Y, and Z values from 0 to 5)
```

Note that `:constants256` does not include the other constants, so if you want to mix both, you need to include `:constants` as well. You may want to explicitly import at least RESET, as in:

```
use Term::ANSIColor 4.00 qw(RESET :constants256);
```

When using the constants, if you don't want to have to remember to add the `, RESET` at the end of each print line, you can set `$Term::ANSIColor::AUTORESET` to a true value. Then, the display mode will automatically be reset if there is no comma after the constant. In other words, with that variable set:

```
print BOLD BLUE "Text\n";
```

will reset the display mode afterward, whereas:

```
print BOLD, BLUE, "Text\n";
```

will not. If you are using background colors, you will probably want to either use *say()* (in newer versions of Perl) or print the newline with a separate print statement to avoid confusing the terminal.

If `$Term::ANSIColor::AUTOLOCAL` is set (see below), it takes precedence over `$Term::ANSIColor::AUTORESET`, and the latter is ignored.

The subroutine interface has the advantage over the constants interface in that only two subroutines are exported into your namespace, versus thirty-eight in the constants interface. On the flip side, the constants interface has the advantage of better compile time error checking, since misspelled names of colors or attributes in calls to *color()* and *colored()* won't be caught until runtime whereas misspelled names of constants will be caught at compile time. So, pollute your namespace with almost two dozen subroutines that you may not even use that often, or risk a silly bug by mistyping an attribute. Your choice, TMTOWTDI after all.

The Color Stack

You can import `:pushpop` and maintain a stack of colors using PUSHCOLOR, POPCOLOR, and LOCALCOLOR. PUSHCOLOR takes the attribute string that starts its argument and pushes it onto a stack of attributes. POPCOLOR removes the top of the stack and restores the previous attributes set by the argument of a prior PUSHCOLOR. LOCALCOLOR surrounds its argument in a PUSHCOLOR and POPCOLOR so that the color resets afterward.

If `$Term::ANSIColor::AUTOLOCAL` is set, each sequence of color constants will be implicitly preceded by LOCALCOLOR. In other words, the following:

```
{
local $Term::ANSIColor::AUTOLOCAL = 1;
print BLUE "Text\n";
}
```

is equivalent to:

```
print LOCALCOLOR BLUE "Text\n";
```

If `$Term::ANSIColor::AUTOLOCAL` is set, it takes precedence over `$Term::ANSIColor::AUTORESET`, and the latter is ignored.

When using PUSHCOLOR, POPCOLOR, and LOCALCOLOR, it's particularly important to not put commas between the constants.

```
print PUSHCOLOR BLUE "Text\n";
```

will correctly push BLUE onto the top of the stack.

```
print PUSHCOLOR, BLUE, "Text\n"; # wrong!
```

will not, and a subsequent pop won't restore the correct attributes. PUSHCOLOR pushes the attributes set by its argument, which is normally a string of color constants. It can't ask the terminal what the current

attributes are.

DIAGNOSTICS

Bad color mapping %s
> (W) The specified color mapping from ANSI_COLORS_ALIASES is not valid and could not be parsed. It was ignored.

Bad escape sequence %s
> (F) You passed an invalid ANSI escape sequence to *uncolor()*.

Bareword "%s" not allowed while "strict subs" in use
> (F) You probably mistyped a constant color name such as:

```
 $Foobar = FOOBAR . "This line should be blue\n";
```

> or:

```
 @Foobar = FOOBAR, "This line should be blue\n";
```

> This will only show up under use strict (another good reason to run under use strict).

Cannot alias standard color %s
> (F) The alias name passed to *coloralias()* matches a standard color name. Standard color names cannot be aliased.

Cannot alias standard color %s in %s
> (W) The same, but in ANSI_COLORS_ALIASES. The color mapping was ignored.

Invalid alias name %s
> (F) You passed an invalid alias name to *coloralias()*. Alias names must consist only of alphanumerics, ., -, and _.

Invalid alias name %s in %s
> (W) You specified an invalid alias name on the left hand of the equal sign in a color mapping in ANSI_COLORS_ALIASES. The color mapping was ignored.

Invalid attribute name %s
> (F) You passed an invalid attribute name to *color()*, *colored()*, or *coloralias()*.

Invalid attribute name %s in %s
> (W) You specified an invalid attribute name on the right hand of the equal sign in a color mapping in ANSI_COLORS_ALIASES. The color mapping was ignored.

Name "%s" used only once: possible typo
> (W) You probably mistyped a constant color name such as:

```
 print FOOBAR "This text is color FOOBAR\n";
```

> It's probably better to always use commas after constant names in order to force the next error.

No comma allowed after filehandle
> (F) You probably mistyped a constant color name such as:

```
 print FOOBAR, "This text is color FOOBAR\n";
```

> Generating this fatal compile error is one of the main advantages of using the constants interface, since you'll immediately know if you mistype a color name.

No name for escape sequence %s
> (F) The ANSI escape sequence passed to *uncolor()* contains escapes which aren't recognized and can't be translated to names.

ENVIRONMENT

ANSI_COLORS_ALIASES
> This environment variable allows the user to specify custom color aliases that will be understood by *color()*, *colored()*, and *colorvalid()*. None of the other functions will be affected, and no new color

constants will be created. The custom colors are aliases for existing color names; no new escape sequences can be introduced. Only alphanumerics, ., _, and - are allowed in alias names.

The format is:

```
ANSI_COLORS_ALIASES='newcolor1=oldcolor1,newcolor2=oldcolor2'
```

Whitespace is ignored.

For example the Solarized <http://ethanschoonover.com/solarized> colors can be mapped with:

```
ANSI_COLORS_ALIASES='\
base00=bright_yellow, on_base00=on_bright_yellow,\
base01=bright_green, on_base01=on_bright_green, \
base02=black, on_base02=on_black, \
base03=bright_black, on_base03=on_bright_black, \
base0=bright_blue, on_base0=on_bright_blue, \
base1=bright_cyan, on_base1=on_bright_cyan, \
base2=white, on_base2=on_white, \
base3=bright_white, on_base3=on_bright_white, \
orange=bright_red, on_orange=on_bright_red, \
violet=bright_magenta,on_violet=on_bright_magenta'
```

This environment variable is read and applied when the Term::ANSIColor module is loaded and is then subsequently ignored. Changes to ANSI_COLORS_ALIASES after the module is loaded will have no effect. See *coloralias()* for an equivalent facility that can be used at runtime.

ANSI_COLORS_DISABLED

If this environment variable is set to a true value, all of the functions defined by this module (*color()*, *colored()*, and all of the constants not previously used in the program) will not output any escape sequences and instead will just return the empty string or pass through the original text as appropriate. This is intended to support easy use of scripts using this module on platforms that don't support ANSI escape sequences.

COMPATIBILITY

Term::ANSIColor was first included with Perl in Perl 5.6.0.

The *uncolor()* function and support for ANSI_COLORS_DISABLED were added in Term::ANSIColor 1.04, included in Perl 5.8.0.

Support for dark was added in Term::ANSIColor 1.08, included in Perl 5.8.4.

The color stack, including the :pushpop import tag, PUSHCOLOR, POPCOLOR, LOCALCOLOR, and the $Term::ANSIColor::AUTOLOCAL variable, was added in Term::ANSIColor 2.00, included in Perl 5.10.1.

colorstrip() was added in Term::ANSIColor 2.01 and *colorvalid()* was added in Term::ANSIColor 2.02, both included in Perl 5.11.0.

Support for colors 8 through 15 (the bright_ variants) was added in Term::ANSIColor 3.00, included in Perl 5.13.3.

Support for italic was added in Term::ANSIColor 3.02, included in Perl 5.17.1.

Support for colors 16 through 256 (the ansi, rgb, and grey colors), the :constants256 import tag, the *coloralias()* function, and support for the ANSI_COLORS_ALIASES environment variable were added in Term::ANSIColor 4.00, included in Perl 5.17.8.

$Term::ANSIColor::AUTOLOCAL was changed to take precedence over $Term::ANSIColor::AUTORESET, rather than the other way around, in Term::ANSIColor 4.00, included in Perl 5.17.8.

RESTRICTIONS

It would be nice if one could leave off the commas around the constants entirely and just say:

```
print BOLD BLUE ON_WHITE "Text\n" RESET;
```

but the syntax of Perl doesn't allow this. You need a comma after the string. (Of course, you may consider it a bug that commas between all the constants aren't required, in which case you may feel free to insert commas unless you're using $Term::ANSIColor::AUTORESET or PUSHCOLOR/POPCOLOR.)

For easier debugging, you may prefer to always use the commas when not setting $Term::ANSIColor::AUTORESET or PUSHCOLOR/POPCOLOR so that you'll get a fatal compile error rather than a warning.

It's not possible to use this module to embed formatting and color attributes using Perl formats. They replace the escape character with a space (as documented in *perlform(1)*), resulting in garbled output from the unrecognized attribute. Even if there were a way around that problem, the format doesn't know that the non-printing escape sequence is zero-length and would incorrectly format the output. For formatted output using color or other attributes, either use *sprintf()* instead or use *formline()* and then add the color or other attributes after formatting and before output.

NOTES

The codes generated by this module are standard terminal control codes, complying with ECMA-048 and ISO 6429 (generally referred to as ''ANSI color'' for the color codes). The non-color control codes (bold, dark, italic, underline, and reverse) are part of the earlier ANSI X3.64 standard for control sequences for video terminals and peripherals.

Note that not all displays are ISO 6429-compliant, or even X3.64-compliant (or are even attempting to be so). This module will not work as expected on displays that do not honor these escape sequences, such as cmd.exe, 4nt.exe, and command.com under either Windows NT or Windows 2000. They may just be ignored, or they may display as an ESC character followed by some apparent garbage.

Jean Delvare provided the following table of different common terminal emulators and their support for the various attributes and others have helped me flesh it out:

```
clear bold faint under blink reverse conceal
-----------------------------------------------------------------------
xterm yes yes no yes yes yes yes
linux yes yes yes bold yes yes no
rxvt yes yes no yes bold/black yes no
dtterm yes yes yes yes reverse yes yes
teraterm yes reverse no yes rev/red yes no
aixterm kinda normal no yes no yes yes
PuTTY yes color no yes no yes no
Windows yes no no no no yes no
Cygwin SSH yes yes no color color color yes
Terminal.app yes yes no yes yes yes yes
```

Windows is Windows telnet, Cygwin SSH is the OpenSSH implementation under Cygwin on Windows NT, and Mac Terminal is the Terminal application in Mac OS X. Where the entry is other than yes or no, that emulator displays the given attribute as something else instead. Note that on an aixterm, clear doesn't reset colors; you have to explicitly set the colors back to what you want. More entries in this table are welcome.

Support for code 3 (italic) is rare and therefore not mentioned in that table. It is not believed to be fully supported by any of the terminals listed, although it's displayed as green in the Linux console, but it is reportedly supported by urxvt.

Note that codes 6 (rapid blink) and 9 (strike-through) are specified in ANSI X3.64 and ECMA-048 but are not commonly supported by most displays and emulators and therefore aren't supported by this module at the present time. ECMA-048 also specifies a large number of other attributes, including a sequence of attributes for font changes, Fraktur characters, double-underlining, framing, circling, and overlining. As none of these attributes are widely supported or useful, they also aren't currently supported by this module.

Most modern X terminal emulators support 256 colors. Known to not support those colors are aterm, rxvt, Terminal.app, and TTY/VC.

AUTHORS

Original idea (using constants) by Zenin, reimplemented using subs by Russ Allbery <rra@cpan.org>, and then combined with the original idea by Russ with input from Zenin. 256-color support is based on work by Kurt Starsinic. Russ Allbery now maintains this module.

PUSHCOLOR, POPCOLOR, and LOCALCOLOR were contributed by openmethods.com voice solutions.

COPYRIGHT AND LICENSE

Copyright 1996 Zenin

Copyright 1996, 1997, 1998, 2000, 2001, 2002, 2005, 2006, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015 Russ Allbery <rra@cpan.org>

Copyright 2012 Kurt Starsinic <kstarsinic@gmail.com>

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

The CPAN module Term::ExtendedColor provides a different and more comprehensive interface for 256-color emulators that may be more convenient. The CPAN module Win32::Console::ANSI provides ANSI color (and other escape sequence) support in the Win32 Console environment.

ECMA-048 is available on-line (at least at the time of this writing) at <http://www.ecma-international.org/publications/standards/Ecma-048.htm>.

ISO 6429 is available from ISO for a charge; the author of this module does not own a copy of it. Since the source material for ISO 6429 was ECMA-048 and the latter is available for free, there seems little reason to obtain the ISO standard.

The 256-color control sequences are documented at <http://invisible-island.net/xterm/ctlseqs/ctlseqs.html> (search for 256-color).

The current version of this module is always available from its web site at <http://www.eyrie.org/~eagle/software/ansicolor/>.
 It is also part of the Perl core distribution as of 5.6.0.