Console Virtual Terminal Sequences

12/30/2021 • 26 minutes to read • Edit Online

Virtual terminal sequences are control character sequences that can control cursor movement, color/font mode, and other operations when written to the output stream. Sequences may also be received on the input stream in response to an output stream query information sequence or as an encoding of user input when the appropriate mode is set.

You can use **GetConsoleMode** and **SetConsoleMode** functions to configure this behavior. A sample of the suggested way to enable virtual terminal behaviors is included at the end of this document.

The behavior of the following sequences is based on the VT100 and derived terminal emulator technologies, most specifically the xterm terminal emulator. More information about terminal sequences can be found at http://vt100.net and at http://invisible-island.net/xterm/ctlseqs/ctlseqs.html.

Output Sequences

The following terminal sequences are intercepted by the console host when written into the output stream, if the ENABLE_VIRTUAL_TERMINAL_PROCESSING flag is set on the screen buffer handle using the **SetConsoleMode** function. Note that the DISABLE_NEWLINE_AUTO_RETURN flag may also be useful in emulating the cursor positioning and scrolling behavior of other terminal emulators in relation to characters written to the final column in any row.

Simple Cursor Positioning

In all of the following descriptions, ESC is always the hexadecimal value 0x1B. No spaces are to be included in terminal sequences. For an example of how these sequences are used in practice, please see the example at the end of this topic.

The following table describes simple escape sequences with a single action command directly after the ESC character. These sequences have no parameters and take effect immediately.

All commands in this table are generally equivalent to calling the **SetConsoleCursorPosition** console API to place the cursor.

Cursor movement will be bounded by the current viewport into the buffer. Scrolling (if available) will not occur.

SEQUENCE	SHORTHAND	BEHAVIOR
ESC M	RI	Reverse Index – Performs the reverse operation of \n, moves cursor up one line, maintains horizontal position, scrolls buffer if necessary*
ESC 7	DECSC	Save Cursor Position in Memory**
ESC 8	DECSR	Restore Cursor Position from Memory**

NOTE

* If there are scroll margins set, RI inside the margins will scroll only the contents of the margins, and leave the viewport unchanged. (See Scrolling Margins)

**There will be no value saved in memory until the first use of the save command. The only way to access the saved value is with the restore command.

Cursor Positioning

The following tables encompass Control Sequence Introducer (CSI) type sequences. All CSI sequences start with ESC (0x1B) followed by [(left bracket, 0x5B) and may contain parameters of variable length to specify more information for each operation. This will be represented by the shorthand <n>. Each table below is grouped by functionality with notes below each table explaining how the group works.

For all parameters, the following rules apply unless otherwise noted:

- <n> represents the distance to move and is an optional parameter
- If <n> is omitted or equals 0, it will be treated as a 1
- <n> cannot be larger than 32,767 (maximum short value)
- <n> cannot be negative

All commands in this section are generally equivalent to calling the **SetConsoleCursorPosition** console API.

Cursor movement will be bounded by the current viewport into the buffer. Scrolling (if available) will not occur.

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [<n> A	CUU	Cursor Up	Cursor up by <n>
ESC [<n> B	CUD	Cursor Down	Cursor down by <n>
ESC [<n> C	CUF	Cursor Forward	Cursor forward (Right) by <n>
ESC [<n> D	CUB	Cursor Backward	Cursor backward (Left) by <n>
ESC [<n> E	CNL	Cursor Next Line	Cursor down <n> lines from current position
ESC [<n> F	CPL	Cursor Previous Line	Cursor up <n> lines from current position
ESC [<n> G	CHA	Cursor Horizontal Absolute	Cursor moves to <n>th position horizontally in the current line
ESC [<n> d	VPA	Vertical Line Position Absolute	Cursor moves to the <n>th position vertically in the current column

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [<y> ; <x> H	CUP	Cursor Position	*Cursor moves to <x>; <y> coordinate within the viewport, where <x> is the column of the <y> line
ESC [<y> ; <x> f	HVP	Horizontal Vertical Position	*Cursor moves to <x>; <y> coordinate within the viewport, where <x> is the column of the <y> line
ESC [s	ANSISYSSC	Save Cursor – Ansi.sys emulation	**With no parameters, performs a save cursor operation like DECSC
ESC [u	ANSISYSSC	Restore Cursor – Ansi.sys emulation	**With no parameters, performs a restore cursor operation like DECRC

NOTE

*<x> and <y> parameters have the same limitations as <n> above. If <x> and <y> are omitted, they will be set to 1;1.

**ANSI.sys historical documentation can be found at https://msdn.microsoft.com/library/cc722862.aspx and is implemented for convenience/compatibility.

Cursor Visibility

The following commands control the visibility of the cursor and its blinking state. The DECTCEM sequences are generally equivalent to calling **SetConsoleCursorInfo** console API to toggle cursor visibility.

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [? 12 h	ATT160	Text Cursor Enable Blinking	Start the cursor blinking
ESC [? 12 l	ATT160	Text Cursor Disable Blinking	Stop blinking the cursor
ESC [? 25 h	DECTCEM	Text Cursor Enable Mode Show	Show the cursor
ESC [? 25 l	DECTCEM	Text Cursor Enable Mode Hide	Hide the cursor

TIP

The enable sequences end in a lowercase H character (`h`) and the disable sequences end in a lowercase L character (`l`).

Viewport Positioning

All commands in this section are generally equivalent to calling **ScrollConsoleScreenBuffer** console API to move the contents of the console buffer.

Caution The command names are misleading. Scroll refers to which direction the text moves during the

operation, not which way the viewport would seem to move.

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [<n> S	SU	Scroll Up	Scroll text up by <n>. Also known as pan down, new lines fill in from the bottom of the screen
ESC [<n> T	SD	Scroll Down	Scroll down by <n>. Also known as pan up, new lines fill in from the top of the screen

The text is moved starting with the line the cursor is on. If the cursor is on the middle row of the viewport, then scroll up would move the bottom half of the viewport, and insert blank lines at the bottom. Scroll down would move the top half of the viewport's rows, and insert new lines at the top.

Also important to note is scroll up and down are also affected by the scrolling margins. Scroll up and down won't affect any lines outside the scrolling margins.

The default value for <n> is 1, and the value can be optionally omitted.

Text Modification

All commands in this section are generally equivalent to calling **FillConsoleOutputCharacter**, **FillConsoleOutputAttribute**, and **ScrollConsoleScreenBuffer** console APIs to modify the text buffer contents.

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [<n> @	ICH	Insert Character	Insert <n> spaces at the current cursor position, shifting all existing text to the right. Text exiting the screen to the right is removed.
ESC [<n> P	DCH	Delete Character	Delete <n> characters at the current cursor position, shifting in space characters from the right edge of the screen.
ESC [<n> X	ECH	Erase Character	Erase <n> characters from the current cursor position by overwriting them with a space character.
ESC [<n> L	IL	Insert Line	Inserts <n> lines into the buffer at the cursor position. The line the cursor is on, and lines below it, will be shifted downwards.
ESC [<n> M	DL	Delete Line	Deletes <n> lines from the buffer, starting with the row the cursor is on.

> **NOTE**
> For IL and DL, only the lines in the scrolling margins (see Scrolling Margins) are affected. If no margins are set, the default margin borders are the current viewport. If lines would be shifted below the margins, they are discarded. When lines are deleted, blank lines are inserted at the bottom of the margins, lines from outside the viewport are never affected.

For each of the sequences, the default value for <n> if it is omitted is 0.

For the following commands, the parameter <n> has 3 valid values:

- 0 erases from the current cursor position (inclusive) to the end of the line/display
- 1 erases from the beginning of the line/display up to and including the current cursor position
- 2 erases the entire line/display

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [<n> J	ED	Erase in Display	Replace all text in the current viewport/screen specified by <n> with space characters
ESC [<n> K	EL	Erase in Line	Replace all text on the line with the cursor specified by <n> with space characters

Text Formatting

All commands in this section are generally equivalent to calling SetConsoleTextAttribute console APIs to adjust the formatting of all future writes to the console output text buffer.

This command is special in that the <n> position below can accept between 0 and 16 parameters separated by semicolons.

When no parameters are specified, it is treated the same as a single 0 parameter.

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [<n> m	SGR	Set Graphics Rendition	Set the format of the screen and text as specified by <n>

The following table of values can be used in <n> to represent different formatting modes.

Formatting modes are applied from left to right. Applying competing formatting options will result in the right-most option taking precedence.

For options that specify colors, the colors will be used as defined in the console color table which can be modified using the SetConsoleScreenBufferInfoEx API. If the table is modified to make the "blue" position in the table display an RGB shade of red, then all calls to **Foreground Blue** will display that red color until otherwise changed.

VALUE	DESCRIPTION	BEHAVIOR
0	Default	Returns all attributes to the default state prior to modification

VALUE	DESCRIPTION	BEHAVIOR
1	Bold/Bright	Applies brightness/intensity flag to foreground color
22	No bold/bright	Removes brightness/intensity flag from foreground color
4	Underline	Adds underline
24	No underline	Removes underline
7	Negative	Swaps foreground and background colors
27	Positive (No negative)	Returns foreground/background to normal
30	Foreground Black	Applies non-bold/bright black to foreground
31	Foreground Red	Applies non-bold/bright red to foreground
32	Foreground Green	Applies non-bold/bright green to foreground
33	Foreground Yellow	Applies non-bold/bright yellow to foreground
34	Foreground Blue	Applies non-bold/bright blue to foreground
35	Foreground Magenta	Applies non-bold/bright magenta to foreground
36	Foreground Cyan	Applies non-bold/bright cyan to foreground
37	Foreground White	Applies non-bold/bright white to foreground
38	Foreground Extended	Applies extended color value to the foreground (see details below)
39	Foreground Default	Applies only the foreground portion of the defaults (see 0)
40	Background Black	Applies non-bold/bright black to background
41	Background Red	Applies non-bold/bright red to background

VALUE	DESCRIPTION	BEHAVIOR
42	Background Green	Applies non-bold/bright green to background
43	Background Yellow	Applies non-bold/bright yellow to background
44	Background Blue	Applies non-bold/bright blue to background
45	Background Magenta	Applies non-bold/bright magenta to background
46	Background Cyan	Applies non-bold/bright cyan to background
47	Background White	Applies non-bold/bright white to background
48	Background Extended	Applies extended color value to the background (see details below)
49	Background Default	Applies only the background portion of the defaults (see 0)
90	Bright Foreground Black	Applies bold/bright black to foreground
91	Bright Foreground Red	Applies bold/bright red to foreground
92	Bright Foreground Green	Applies bold/bright green to foreground
93	Bright Foreground Yellow	Applies bold/bright yellow to foreground
94	Bright Foreground Blue	Applies bold/bright blue to foreground
95	Bright Foreground Magenta	Applies bold/bright magenta to foreground
96	Bright Foreground Cyan	Applies bold/bright cyan to foreground
97	Bright Foreground White	Applies bold/bright white to foreground
100	Bright Background Black	Applies bold/bright black to background
101	Bright Background Red	Applies bold/bright red to background
102	Bright Background Green	Applies bold/bright green to background

VALUE	DESCRIPTION	BEHAVIOR
103	Bright Background Yellow	Applies bold/bright yellow to background
104	Bright Background Blue	Applies bold/bright blue to background
105	Bright Background Magenta	Applies bold/bright magenta to background
106	Bright Background Cyan	Applies bold/bright cyan to background
107	Bright Background White	Applies bold/bright white to background

Extended Colors

Some virtual terminal emulators support a palette of colors greater than the 16 colors provided by the Windows Console. For these extended colors, the Windows Console will choose the nearest appropriate color from the existing 16 color table for display. Unlike typical SGR values above, the extended values will consume additional parameters after the initial indicator according to the table below.

SGR SUBSEQUENCE	DESCRIPTION
38 ; 2 ; <r> ; <g> ; <b>	Set foreground color to RGB value specified in <r>, <g>, <b> parameters*
48 ; 2 ; <r> ; <g> ; <b>	Set background color to RGB value specified in <r>, <g>, <b> parameters*
38 ; 5 ; <s>	Set foreground color to <s> index in 88 or 256 color table*
48 ; 5 ; <s>	Set background color to <s> index in 88 or 256 color table*

*The 88 and 256 color palettes maintained internally for comparison are based from the xterm terminal emulator. The comparison/rounding tables cannot be modified at this time.

Screen Colors

The following command allows the application to set the screen colors palette values to any RGB value.

The RGB values should be hexadecimal values between `0` and `ff`, and separated by the forward-slash character (e.g. `rgb:1/24/86`).

Note that this sequence is an OSC "Operating system command" sequence, and not a CSI like many of the other sequences listed, and as such start with "\x1b]", not "\x1b[". As OSC sequences, they are ended with a *String Terminator* represented as `<ST>` and transmitted with `ESC \` (`0x1B 0x5C`). `BEL` (`0x7`) may be used instead as the terminator, but the longer form is preferred.

SEQUENCE	DESCRIPTION	BEHAVIOR

SEQUENCE	DESCRIPTION	BEHAVIOR
ESC] 4 ; <i> ; rgb : <r> / <g> / <b> <ST>	Modify Screen Colors	Sets the screen color palette index <i> to the RGB values specified in <r>, <g>, <b>

Mode Changes

These are sequences that control the input modes. There are two different sets of input modes, the Cursor Keys Mode and the Keypad Keys Mode. The Cursor Keys Mode controls the sequences that are emitted by the arrow keys as well as Home and End, while the Keypad Keys Mode controls the sequences emitted by the keys on the numpad primarily, as well as the function keys.

Each of these modes are simple boolean settings – the Cursor Keys Mode is either Normal (default) or Application, and the Keypad Keys Mode is either Numeric (default) or Application.

See the Cursor Keys and Numpad & Function Keys sections for the sequences emitted in these modes.

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC =	DECKPAM	Enable Keypad Application Mode	Keypad keys will emit their Application Mode sequences.
ESC >	DECKPNM	Enable Keypad Numeric Mode	Keypad keys will emit their Numeric Mode sequences.
ESC [? 1 h	DECCKM	Enable Cursor Keys Application Mode	Keypad keys will emit their Application Mode sequences.
ESC [? 1 l	DECCKM	Disable Cursor Keys Application Mode (use Normal Mode)	Keypad keys will emit their Numeric Mode sequences.

Query State

All commands in this section are generally equivalent to calling Get* console APIs to retrieve status information about the current console buffer state.

> **NOTE**
> These queries will emit their responses into the console input stream immediately after being recognized on the output stream while ENABLE_VIRTUAL_TERMINAL_PROCESSING is set. The ENABLE_VIRTUAL_TERMINAL_INPUT flag does not apply to query commands as it is assumed that an application making the query will always want to receive the reply.

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [6 n	DECXCPR	Report Cursor Position	Emit the cursor position as: ESC [<r> ; <c> R Where <r> = cursor row and <c> = cursor column

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [0 c	DA	Device Attributes	Report the terminal identity. Will emit “\x1b[?1;0c”, indicating "VT101 with No Options".

Tabs

While the windows console traditionally expects tabs to be exclusively eight characters wide, *nix applications utilizing certain sequences can manipulate where the tab stops are within the console windows to optimize cursor movement by the application.

The following sequences allow an application to set the tab stop locations within the console window, remove them, and navigate between them.

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC H	HTS	Horizontal Tab Set	Sets a tab stop in the current column the cursor is in.
ESC [<n> I	CHT	Cursor Horizontal (Forward) Tab	Advance the cursor to the next column (in the same row) with a tab stop. If there are no more tab stops, move to the last column in the row. If the cursor is in the last column, move to the first column of the next row.
ESC [<n> Z	CBT	Cursor Backwards Tab	Move the cursor to the previous column (in the same row) with a tab stop. If there are no more tab stops, moves the cursor to the first column. If the cursor is in the first column, doesn’t move the cursor.
ESC [0 g	TBC	Tab Clear (current column)	Clears the tab stop in the current column, if there is one. Otherwise does nothing.
ESC [3 g	TBC	Tab Clear (all columns)	Clears all currently set tab stops.

- For both CHT and CBT, <n> is an optional parameter that (default=1) indicating how many times to advance the cursor in the specified direction.
- If there are no tab stops set via HTS, CHT and CBT will treat the first and last columns of the window as the only two tab stops.
- Using HTS to set a tab stop will also cause the console to navigate to the next tab stop on the output of a TAB (0x09, ‘\t’) character, in the same manner as CHT.

Designate Character Set

The following sequences allow a program to change the active character set mapping. This allows a program to emit 7-bit ASCII characters, but have them displayed as other glyphs on the terminal screen itself. Currently, the only two supported character sets are ASCII (default) and the DEC Special Graphics Character Set. See http://vt100.net/docs/vt220-rm/table2-4.html for a listing of all of the characters represented by the DEC Special Graphics Character Set.

SEQUENCE	DESCRIPTION	BEHAVIOR
ESC (0	Designate Character Set – DEC Line Drawing	Enables DEC Line Drawing Mode
ESC (B	Designate Character Set – US ASCII	Enables ASCII Mode (Default)

Notably, the DEC Line Drawing mode is used for drawing borders in console applications. The following table shows what ASCII character maps to which line drawing character.

HEX	ASCII	DEC LINE DRAWING
0x6a	j	┘
0x6b	k	┐
0x6c	l	┌
0x6d	m	└
0x6e	n	┼
0x71	q	─
0x74	t	├
0x75	u	┤
0x76	v	┴
0x77	w	┬
0x78	x	│

Scrolling Margins

The following sequences allow a program to configure the “scrolling region” of the screen that is affected by scrolling operations. This is a subset of the rows that are adjusted when the screen would otherwise scroll, for example, on a ‘\n’ or RI. These margins also affect the rows modified by Insert Line (IL) and Delete Line (DL), Scroll Up (SU) and Scroll Down (SD).

The scrolling margins can be especially useful for having a portion of the screen that doesn’t scroll when the rest of the screen is filled, such as having a title bar at the top or a status bar at the bottom of your application.

For DECSTBM, there are two optional parameters, <t> and <b>, which are used to specify the rows that represent the top and bottom lines of the scroll region, inclusive. If the parameters are omitted, <t> defaults to 1

and <b> defaults to the current viewport height.

Scrolling margins are per-buffer, so importantly, the Alternate Buffer and Main Buffer maintain separate scrolling margins settings (so a full screen application in the alternate buffer will not poison the main buffer's margins).

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [<t> ; <b> r	DECSTBM	Set Scrolling Region	Sets the VT scrolling margins of the viewport.

Window Title

The following commands allows the application to set the title of the console window to the given <string> parameter. The string must be less than 255 characters to be accepted. This is equivalent to calling SetConsoleTitle with the given string.

Note that these sequences are OSC "Operating system command" sequences, and not a CSI like many of the other sequences listed, and as such starts with "\x1b]", not "\x1b[". As OSC sequences, they are ended with a *String Terminator* represented as `<ST>` and transmitted with `ESC \` (`0x1B 0x5C`). `BEL` (`0x7`) may be used instead as the terminator, but the longer form is preferred.

SEQUENCE	DESCRIPTION	BEHAVIOR
ESC] 0 ; <string> <ST>	Set Window Title	Sets the console window's title to <string>.
ESC] 2 ; <string> <ST>	Set Window Title	Sets the console window's title to <string>.

The terminating character here is the "Bell" character, '\x07'

Alternate Screen Buffer

*Nix style applications often utilize an alternate screen buffer, so that they can modify the entire contents of the buffer, without affecting the application that started them. The alternate buffer is exactly the dimensions of the window, without any scrollback region.

For an example of this behavior, consider when vim is launched from bash. Vim uses the entirety of the screen to edit the file, then returning to bash leaves the original buffer unchanged.

SEQUENCE	DESCRIPTION	BEHAVIOR
ESC [? 1 0 4 9 h	Use Alternate Screen Buffer	Switches to a new alternate screen buffer.
ESC [? 1 0 4 9 l	Use Main Screen Buffer	Switches to the main buffer.

Window Width

The following sequences can be used to control the width of the console window. They are roughly equivalent to the calling the SetConsoleScreenBufferInfoEx console API to set the window width.

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [? 3 h	DECCOLM	Set Number of Columns to 132	Sets the console width to 132 columns wide.
ESC [? 3 l	DECCOLM	Set Number of Columns to 80	Sets the console width to 80 columns wide.

Soft Reset

The following sequence can be used to reset certain properties to their default values.The following properties are reset to the following default values (also listed are the sequences that control those properties):

- Cursor visibility: visible (DECTEM)
- Numeric Keypad: Numeric Mode (DECNKM)
- Cursor Keys Mode: Normal Mode (DECCKM)
- Top and Bottom Margins: Top=1, Bottom=Console height (DECSTBM)
- Character Set: US ASCII
- Graphics Rendition: Default/Off (SGR)
- Save cursor state: Home position (0,0) (DECSC)

SEQUENCE	CODE	DESCRIPTION	BEHAVIOR
ESC [! p	DECSTR	Soft Reset	Reset certain terminal settings to their defaults.

Input Sequences

The following terminal sequences are emitted by the console host on the input stream if the ENABLE_VIRTUAL_TERMINAL_INPUT flag is set on the input buffer handle using the SetConsoleMode flag.

There are two internal modes that control which sequences are emitted for the given input keys, the Cursor Keys Mode and the Keypad Keys Mode. These are described in the Mode Changes section.

Cursor Keys

KEY	NORMAL MODE	APPLICATION MODE
Up Arrow	ESC [A	ESC O A
Down Arrow	ESC [B	ESC O B
Right Arrow	ESC [C	ESC O C
Left Arrow	ESC [D	ESC O D
Home	ESC [H	ESC O H
End	ESC [F	ESC O F

Additionally, if Ctrl is pressed with any of these keys, the following sequences are emitted instead, regardless of the Cursor Keys Mode:

KEY	ANY MODE
Ctrl + Up Arrow	ESC [1 ; 5 A
Ctrl + Down Arrow	ESC [1 ; 5 B
Ctrl + Right Arrow	ESC [1 ; 5 C
Ctrl + Left Arrow	ESC [1 ; 5 D

Numpad & Function Keys

KEY	SEQUENCE
Backspace	0x7f (DEL)
Pause	0x1a (SUB)
Escape	0x1b (ESC)
Insert	ESC [2 ~
Delete	ESC [3 ~
Page Up	ESC [5 ~
Page Down	ESC [6 ~
F1	ESC O P
F2	ESC O Q
F3	ESC O R
F4	ESC O S
F5	ESC [1 5 ~
F6	ESC [1 7 ~
F7	ESC [1 8 ~
F8	ESC [1 9 ~
F9	ESC [2 0 ~
F10	ESC [2 1 ~
F11	ESC [2 3 ~
F12	ESC [2 4 ~

Modifiers

Alt is treated by prefixing the sequence with an escape: ESC <c> where <c> is the character passed by the operating system. Alt+Ctrl is handled the same way except that the operating system will have pre-shifted the <c> key to the appropriate control character which will be relayed to the application.

Ctrl is generally passed through exactly as received from the system. This is typically a single character shifted down into the control character reserved space (0x0-0x1f). For example, Ctrl+@ (0x40) becomes NUL (0x00), Ctrl+[(0x5b) becomes ESC (0x1b), etc. A few Ctrl key combinations are treated specially according to the following table:

KEY	SEQUENCE
Ctrl + Space	0x00 (NUL)
Ctrl + Up Arrow	ESC [1 ; 5 A
Ctrl + Down Arrow	ESC [1 ; 5 B
Ctrl + Right Arrow	ESC [1 ; 5 C
Ctrl + Left Arrow	ESC [1 ; 5 D

> **NOTE**
> Left `Ctrl` + Right `Alt` is treated as AltGr. When both are seen together, they will be stripped and the Unicode value of the character presented by the system will be passed into the target. The system will pre-translate AltGr values according to the current system input settings.

Samples

Example of SGR terminal sequences

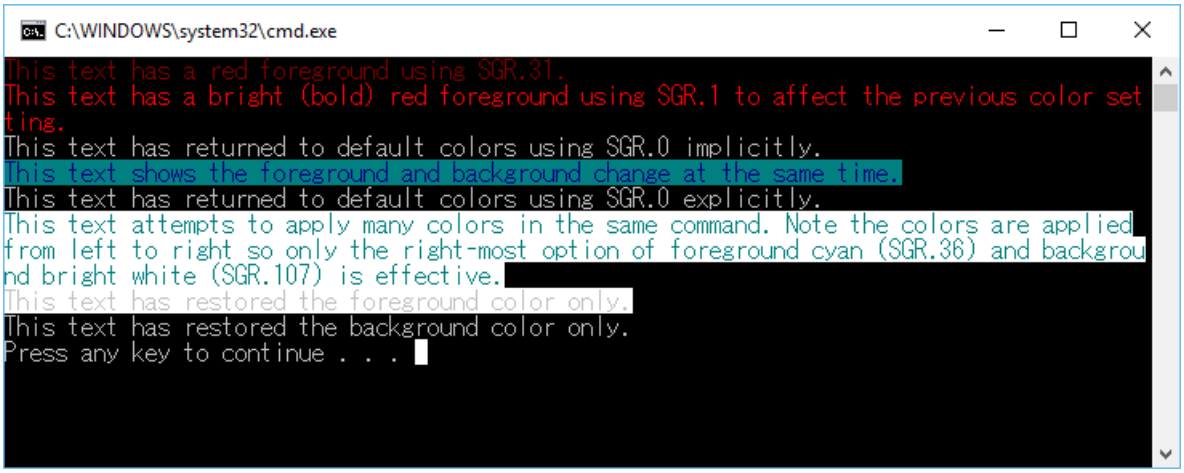
The following code provides several examples of text formatting.

```
#include <stdio.h>
#include <wchar.h>
#include <windows.h>

int main()
{
    // Set output mode to handle virtual terminal sequences
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hOut == INVALID_HANDLE_VALUE)
    {
        return GetLastError();
    }

    DWORD dwMode = 0;
    if (!GetConsoleMode(hOut, &dwMode))
    {
        return GetLastError();
    }

    dwMode |= ENABLE_VIRTUAL_TERMINAL_PROCESSING;
    if (!SetConsoleMode(hOut, dwMode))
    {
        return GetLastError();
    }

    // Try some Set Graphics Rendition (SGR) terminal escape sequences
    wprintf(L"\x1b[31mThis text has a red foreground using SGR.31.\r\n");
    wprintf(L"\x1b[1mThis text has a bright (bold) red foreground using SGR.1 to affect the previous color
setting.\r\n");
    wprintf(L"\x1b[mThis text has returned to default colors using SGR.0 implicitly.\r\n");
    wprintf(L"\x1b[34;46mThis text shows the foreground and background change at the same time.\r\n");
    wprintf(L"\x1b[0mThis text has returned to default colors using SGR.0 explicitly.\r\n");
    wprintf(L"\x1b[31;32;33;34;35;36;101;102;103;104;105;106;107mThis text attempts to apply many colors in
the same command. Note the colors are applied from left to right so only the right-most option of foreground
cyan (SGR.36) and background bright white (SGR.107) is effective.\r\n");
    wprintf(L"\x1b[39mThis text has restored the foreground color only.\r\n");
    wprintf(L"\x1b[49mThis text has restored the background color only.\r\n");

    return 0;
}
```

NOTE

In the previous example, the string '`\x1b[31m`' is the implementation of **ESC [<n> m** with <n> being 31.

The following graphic shows the output of the previous code example.



Example of Enabling Virtual Terminal Processing

The following code provides an example of the recommended way to enable virtual terminal processing for an application. The intent of the sample is to demonstrate:

1. The existing mode should always be retrieved via GetConsoleMode and analyzed before being set with SetConsoleMode.
2. Checking whether SetConsoleMode returns `0` and GetLastError returns ERROR_INVALID_PARAMETER is the current mechanism to determine when running on a down-level system. An application receiving ERROR_INVALID_PARAMETER with one of the newer console mode flags in the bit field should gracefully degrade behavior and try again.

```
#include <stdio.h>
#include <wchar.h>
#include <windows.h>

int main()
{
    // Set output mode to handle virtual terminal sequences
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hOut == INVALID_HANDLE_VALUE)
    {
        return false;
    }
    HANDLE hIn = GetStdHandle(STD_INPUT_HANDLE);
    if (hIn == INVALID_HANDLE_VALUE)
    {
        return false;
    }

    DWORD dwOriginalOutMode = 0;
    DWORD dwOriginalInMode = 0;
    if (!GetConsoleMode(hOut, &dwOriginalOutMode))
    {
        return false;
    }
    if (!GetConsoleMode(hIn, &dwOriginalInMode))
    {
        return false;
    }

    DWORD dwRequestedOutModes = ENABLE_VIRTUAL_TERMINAL_PROCESSING | DISABLE_NEWLINE_AUTO_RETURN;
    DWORD dwRequestedInModes = ENABLE_VIRTUAL_TERMINAL_INPUT;

    DWORD dwOutMode = dwOriginalOutMode | dwRequestedOutModes;
    if (!SetConsoleMode(hOut, dwOutMode))
    {
        // we failed to set both modes, try to step down mode gracefully.
        dwRequestedOutModes = ENABLE_VIRTUAL_TERMINAL_PROCESSING;
        dwOutMode = dwOriginalOutMode | dwRequestedOutModes;
        if (!SetConsoleMode(hOut, dwOutMode))
        {
            // Failed to set any VT mode, can't do anything here.
            return -1;
        }
    }

    DWORD dwInMode = dwOriginalInMode | ENABLE_VIRTUAL_TERMINAL_INPUT;
    if (!SetConsoleMode(hIn, dwInMode))
    {
        // Failed to set VT input mode, can't do anything here.
        return -1;
    }

    return 0;
}
```

Example of Select Anniversary Update Features

The following example is intended to be a more robust example of code using a variety of escape sequences to manipulate the buffer, with an emphasis on the features added in the Anniversary Update for Windows 10.

This example makes use of the alternate screen buffer, manipulating tab stops, setting scrolling margins, and changing the character set.

```
// System headers
#include <windows.h>
```

```
// Standard library C-style
#include <wchar.h>
#include <stdlib.h>
#include <stdio.h>

#define ESC "\x1b"
#define CSI "\x1b["

bool EnableVTMode()
{
    // Set output mode to handle virtual terminal sequences
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hOut == INVALID_HANDLE_VALUE)
    {
        return false;
    }

    DWORD dwMode = 0;
    if (!GetConsoleMode(hOut, &dwMode))
    {
        return false;
    }

    dwMode |= ENABLE_VIRTUAL_TERMINAL_PROCESSING;
    if (!SetConsoleMode(hOut, dwMode))
    {
        return false;
    }
    return true;
}

void PrintVerticalBorder()
{
    printf(ESC "(0"); // Enter Line drawing mode
    printf(CSI "104;93m"); // bright yellow on bright blue
    printf("x"); // in line drawing mode, \x78 -> \u2502 "Vertical Bar"
    printf(CSI "0m"); // restore color
    printf(ESC "(B"); // exit line drawing mode
}

void PrintHorizontalBorder(COORD const Size, bool fIsTop)
{
    printf(ESC "(0"); // Enter Line drawing mode
    printf(CSI "104;93m"); // Make the border bright yellow on bright blue
    printf(fIsTop ? "l" : "m"); // print left corner

    for (int i = 1; i < Size.X - 1; i++)
        printf("q"); // in line drawing mode, \x71 -> \u2500 "HORIZONTAL SCAN LINE-5"

    printf(fIsTop ? "k" : "j"); // print right corner
    printf(CSI "0m");
    printf(ESC "(B"); // exit line drawing mode
}

void PrintStatusLine(const char* const pszMessage, COORD const Size)
{
    printf(CSI "%d;1H", Size.Y);
    printf(CSI "K"); // clear the line
    printf(pszMessage);
}

int __cdecl wmain(int argc, WCHAR* argv[])
{
    argc; // unused
    argv; // unused
    //First, enable VT mode
    bool fSuccess = EnableVTMode();
    if (!fSuccess)
    {
```

```
{
    printf("Unable to enter VT processing mode. Quitting.\n");
    return -1;
}
HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
if (hOut == INVALID_HANDLE_VALUE)
{
    printf("Couldn't get the console handle. Quitting.\n");
    return -1;
}

CONSOLE_SCREEN_BUFFER_INFO ScreenBufferInfo;
GetConsoleScreenBufferInfo(hOut, &ScreenBufferInfo);
COORD Size;
Size.X = ScreenBufferInfo.srWindow.Right - ScreenBufferInfo.srWindow.Left + 1;
Size.Y = ScreenBufferInfo.srWindow.Bottom - ScreenBufferInfo.srWindow.Top + 1;

// Enter the alternate buffer
printf(CSI "?1049h");

// Clear screen, tab stops, set, stop at columns 16, 32
printf(CSI "1;1H");
printf(CSI "2J"); // Clear screen

int iNumTabStops = 4; // (0, 20, 40, width)
printf(CSI "3g"); // clear all tab stops
printf(CSI "1;20H"); // Move to column 20
printf(ESC "H"); // set a tab stop

printf(CSI "1;40H"); // Move to column 40
printf(ESC "H"); // set a tab stop

// Set scrolling margins to 3, h-2
printf(CSI "3;%dr", Size.Y - 2);
int iNumLines = Size.Y - 4;

printf(CSI "1;1H");
printf(CSI "102;30m");
printf("Windows 10 Anniversary Update - VT Example");
printf(CSI "0m");

// Print a top border - Yellow
printf(CSI "2;1H");
PrintHorizontalBorder(Size, true);

// // Print a bottom border
printf(CSI "%d;1H", Size.Y - 1);
PrintHorizontalBorder(Size, false);

wchar_t wch;

// draw columns
printf(CSI "3;1H");
int line = 0;
for (line = 0; line < iNumLines * iNumTabStops; line++)
{
    PrintVerticalBorder();
    if (line + 1 != iNumLines * iNumTabStops) // don't advance to next line if this is the last line
        printf("\t"); // advance to next tab stop

}

PrintStatusLine("Press any key to see text printed between tab stops.", Size);
wch = _getwch();

// Fill columns with output
printf(CSI "3;1H");
for (line = 0; line < iNumLines; line++)
{
```

```
        int tab = 0;
        for (tab = 0; tab < iNumTabStops - 1; tab++)
        {
            PrintVerticalBorder();
            printf("line=%d", line);
            printf("\t"); // advance to next tab stop
        }
        PrintVerticalBorder();// print border at right side
        if (line + 1 != iNumLines)
            printf("\t"); // advance to next tab stop, (on the next line)
    }

    PrintStatusLine("Press any key to demonstrate scroll margins", Size);
    wch = _getwch();

    printf(CSI "3;1H");
    for (line = 0; line < iNumLines * 2; line++)
    {
        printf(CSI "K"); // clear the line
        int tab = 0;
        for (tab = 0; tab < iNumTabStops - 1; tab++)
        {
            PrintVerticalBorder();
            printf("line=%d", line);
            printf("\t"); // advance to next tab stop
        }
        PrintVerticalBorder(); // print border at right side
        if (line + 1 != iNumLines * 2)
        {
            printf("\n"); //Advance to next line. If we're at the bottom of the margins, the text will
scroll.
            printf("\r"); //return to first col in buffer
        }
    }

    PrintStatusLine("Press any key to exit", Size);
    wch = _getwch();

    // Exit the alternate buffer
    printf(CSI "?1049l");

}
```