Contents

- Home
  - Ecosystem Roadmap
  - Definitions
  - Console APIs vs. Virtual Terminal
- About Character Mode Applications
  - Input And Output Methods
  - Console Code Pages
  - Console Control Handlers
  - Console Aliases
  - Console Buffer Security and Access Rights
  - Console Application Issues
- About Consoles
  - Creation of a Console
  - Attaching to a Console
  - Closing a Console
  - Console Handles
  - Console Input Buffer
  - Console Screen Buffers
  - Console Modes
  - Console Process Groups
  - Window And Screen Buffer Size
  - Console Selection
  - About Legacy Console Mode
  - About Pseudoconsoles
- Console Developer's guide & API Reference
  - Using The Console API
    - High Level Console Input And Output Functions
    - Using The High Level Input And Output Functions
    - High Level Console Modes

- High Level Console I/O
- Low Level Console Input Functions
- Low Level Console Output Functions
- Low Level Console I/O
- Low Level Console Modes
- Reading And Writing Blocks Of Characters And Attributes
- Reading Input Buffer Events
- Clearing the screen
- Scrolling a Screen Buffer
- Scrolling a Screen Buffer's Contents
- Scrolling a Screen Buffer's Window
- Ctrl C And Ctrl Break Signals
- Ctrl Close Signal
- Registering a Control Handler Function
- Console Virtual Terminal Sequences
- Creating a Pseudoconsole Session

Console API Functions

- AddConsoleAlias
- AllocConsole
- AttachConsole
- ClosePseudoConsole
- CreateConsoleScreenBuffer
- CreatePseudoConsole
- FillConsoleOutputAttribute
- FillConsoleOutputCharacter
- FlushConsoleInputBuffer
- FreeConsole
- GenerateConsoleCtrlEvent
- GetConsoleAlias
- GetConsoleAliases
- GetConsoleAliasesLength
- GetConsoleAliasExes

GetConsoleAliasExesLength

GetConsoleCP

GetConsoleCursorInfo

GetConsoleDisplayMode

GetConsoleFontSize

GetConsoleHistoryInfo

GetConsoleMode

GetConsoleOriginalTitle

GetConsoleOutputCP

GetConsoleProcessList

GetConsoleScreenBufferInfo

GetConsoleScreenBufferInfoEx

GetConsoleSelectionInfo

GetConsoleTitle

GetConsoleWindow

GetCurrentConsoleFont

GetCurrentConsoleFontEx

GetLargestConsoleWindowSize

GetNumberOfConsoleInputEvents

GetNumberOfConsoleMouseButtons

GetStdHandle

HandlerRoutine

PeekConsoleInput

ReadConsole

ReadConsoleInput

ReadConsoleOutput

ReadConsoleOutputAttribute

ReadConsoleOutputCharacter

ResizePseudoConsole

ScrollConsoleScreenBuffer

SetConsoleActiveScreenBuffer

SetConsoleCP

- SetConsoleCtrlHandler
- SetConsoleCursorInfo
- SetConsoleCursorPosition
- SetConsoleDisplayMode
- SetConsoleHistoryInfo
- SetConsoleMode
- SetConsoleOutputCP
- SetConsoleScreenBufferInfoEx
- SetConsoleScreenBufferSize
- SetConsoleTextAttribute
- SetConsoleTitle
- SetConsoleWindowInfo
- SetCurrentConsoleFontEx
- SetStdHandle
- WriteConsole
- WriteConsoleInput
- WriteConsoleOutput
- WriteConsoleOutputAttribute
- WriteConsoleOutputCharacter

Console API Structures

- CONSOLE_HISTORY_INFO structure
- CONSOLE_READCONSOLE_CONTROL structure
- CONSOLE_SELECTION_INFO structure
- CONSOLE_CURSOR_INFO structure
- CONSOLE_FONT_INFO structure
- CONSOLE_FONT_INFOEX structure
- CONSOLE_SCREEN_BUFFER_INFO structure
- CONSOLE_SCREEN_BUFFER_INFOEX structure
- CHAR_INFO structure
- COORD structure
- SMALL_RECT structure
- INPUT_RECORD structure

KEY_EVENT_RECORD structure
MENU_EVENT_RECORD structure
MOUSE_EVENT_RECORD structure
FOCUS_EVENT_RECORD structure
WINDOW_BUFFER_SIZE_RECORD structure
Console API Winevents

Welcome to the Windows Console documentation!

5/18/2021 • 2 minutes to read • Edit Online

In the sections on the left of this page, you'll find information about the concepts, APIs and related functions, structures, etc. through which you can programmatically control and interact with the Windows Console.

Windows Console and Terminal Ecosystem Roadmap

5/18/2021 • 9 minutes to read • Edit Online

This document is a high-level roadmap of the Windows Console and Windows Terminal products. It covers:

- How Windows Console and Windows Terminal fit into the ecosystem of command-line applications across Windows and other operating systems.
- A history and future roadmap of the products, features, and strategies that are part of building the platform, as well as building for this platform.

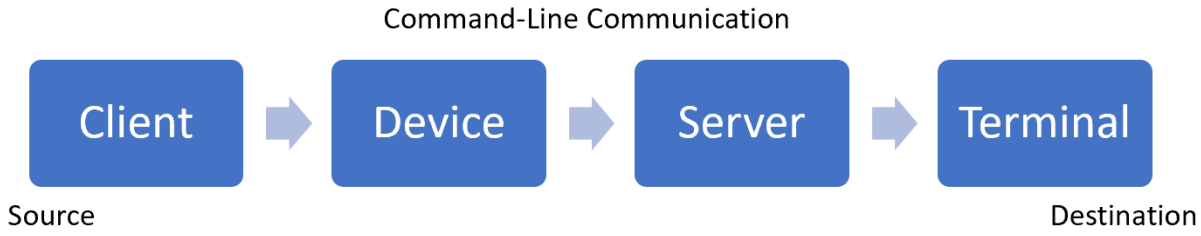
The focus of the current console/terminal era at Microsoft is to bring a first-class terminal experience directly to developers on the Windows platform and to phase out classic Windows Console APIs, replacing them with virtual terminal sequences utilizing pseudoconsole. **Windows Terminal** showcases this transition into a first-class experience, inviting open source collaboration from the developer community, supporting a full spectrum of mixing and matching of client command-line and terminal hosting applications, and unifying the Windows ecosystem with all other platforms.

Definitions

It is recommended to familiarize yourself with the definitions of common terminology used in this space before proceeding. Common terminology includes: Command Line (or Console) applications, standard handles (`STDIN`, `STDOUT`, `STDERR`), TTY and PTY devices, clients and servers, console subsystem, console host, pseudoconsole, and terminal.

Architecture

The general architecture of the system is in four parts: client, device, server, and terminal.



Client

The client is a command-line application that uses a text-based interface to enable the user to enter commands (rather than a mouse-based user interface), returning a text representation of the result. On Windows, the Console API provides a communications layer between the client and the device. (This can also be a standard console handle with device control APIs).

Device

The device is an intermediate message-handling communications layer between two processes, the client and the server. On Windows, this is the console driver. On other platforms, it is the TTY or PTY device. Other devices like files, pipes, and sockets may be used as this communication channel if the entire transaction is in plain text or contains virtual terminal sequences, but not with Windows Console APIs.

Server

The server interprets the requested API calls or messages from the client. On Windows in the classic operating mode, the server also creates a user interface to present the output to the screen. The server additionally collects input to send back in response messages to the client, via the driver, like a terminal bundled in the same module. Using pseudoconsole mode, it instead is only a translator to present this information in virtual terminal sequences to an attached terminal.

Terminal

The terminal is the final layer providing graphical display and interactivity services to the user. It is responsible for capturing input and encoding it as virtual terminal sequences, which eventually reach the client's `STDIN`. It will also receive and decode the *virtual terminal sequences* that it receives back from the client's `STDOUT` for presentation on the screen.

Further connections

As an addendum, further connections can be performed by chaining applications that serve multiple roles into one of the endpoints. For instance, an SSH session has two roles: it is a **terminal** for the command-line application running on one device, but it forwards all received information on to a **client** role on another device. This chaining can occur indefinitely across devices and contexts offering broad scenario flexibility.

On non-Windows platforms, the **server** and **terminal** roles are a single unit because there is no need for a translation compatibility layer between an API set and virtual terminal sequences.

Microsoft products

All of the Microsoft Windows command-line products are now available on GitHub in an open source repository, microsoft/terminal.

Windows Console Host

This is the traditional Windows user-interface for command-line applications. It handles all console API servicing called from any attached command-line application. Windows Console also handles the graphical user interface (GUI) representation on behalf of all of those applications. It is found in the system directory as `conhost.exe`, or `openconsole.exe` in its open source form. It comes with the Windows operating system. It can also be found in other Microsoft products built from the open source repository for a more up-to-date implementation of the pseudoconsole infrastructure. Per the definitions above, it operates in either a combined server-and-terminal role traditionally or a server-only role through the preferred *pseudoconsole* infrastructure.

Windows Terminal

This is the new Windows interface for command-line applications. Windows Terminal serves as a first-party example of using the pseudoconsole to separate the concerns between API servicing and text-based application interfacing, much like all non-Windows platforms.

Windows Terminal is the flagship text-mode user interface for Windows. It demonstrates the capabilities of the ecosystem and is driving Windows development toward unifying with other platforms. Windows Terminal is also an example of how to build a robust and complex modern application that spans the history and gamut of Windows APIs and frameworks. Per the definitions above, this product operates in a terminal role.

Major historical milestones

The major historical milestones for the console subsystem are broken into implementation prior to 2014 and then moves into an overview of work performed since 2014, when the renewed focus on the command-line was formed in the Windows 10 era.

Initial Implementation

[1989-1990s] The initial console host system was implemented as an emulation of the DOS environment within the Windows operating system. Its code is entangled and cooperative with the Command Prompt,

`cmd.exe`, that is a representation of that DOS environment. The console host system code shares responsibilities and privileges with the Command Prompt interpreter/shell. It also provides a base level of services for other command-line utilities to perform services in a CMD-like manner.

DBCS for CJK

[1997-1999] Around this time, DBCS support ("Double-byte character set") is introduced to support CJK (Chinese, Japanese, and Korean) markets. This effort results in a bifurcation of many of the writing and reading methods inside the console to provide both "western" versions to deal with single-byte characters as well as an alternative representation for "eastern" versions where two bytes are required to represent the vast array of characters. This bifurcation included the expanding representation of a cell in the console environment to be either 1 or 2 cells wide, where 1 cell is narrow (taller than it is wide) and 2 cells is wide, full-width, or otherwise a square in which typical Chinese, Japanese, and Korean ideographs can be inscribed.

Security/Isolation

[2005-2009] With the console subsystem experience running inside the critical system process, `csrss.exe`, connecting assorted client applications, at varying access levels, to a single super-critical and privileged process was noticed as particularly dangerous. In this era, the console subsystem was split into client, driver, and server applications. Each application could run in their own context, reducing the responsibilities and privilege in each. This isolation increased the general robustness of the system, as any failure in the console subsystem no longer affected other critical process functionality.

User Experience Improvements

[2014-2016] After a long time of generally scattered maintenance of the console subsystem by assorted teams across the organization, a new developer-focused team was formed to own and drive improvements in the console. Improvements during this time included: line selection, smooth window resizing, reflowing text, copy and paste, high DPI support, and a focus on Unicode, including the convergence of the split between "western" and "eastern" storage and stream manipulation algorithms.

Virtual Terminal client

[2015-2017] With the arrival of the Windows Subsystem for Linux, Microsoft efforts to improve the experience of Docker on Windows, and the adoption of OpenSSH as the premier command-line remote execution technology, the initial implementations of virtual terminal sequences were introduced into the console host. This allowed the existing console to act as the terminal, attached directly to those Linux-native applications in their respective environments, rendering graphical and text attributes to the display and returning user input in the appropriate dialect.

Virtual Terminal server

[2018] Over the past twenty years, third-party alternatives for the inbox console host were created to offer additional developer productivity, prominently centered in rich customizations and tabbed interfaces. These applications still needed to run and hide the console host window. They attach as a secondary "client" application to scrape out buffer information in polling loops as the primary command-line client application operated. Their goal was to be a terminal, like on other platforms, but in the Windows world where terminals were not replaceable.

In this time period, the pseudoconsole infrastructure was introduced. Pseudoconsole permits any application to launch the console host in a non-interactive mode and become the final terminal interface for the user. The main limitation in this effort was the continued compatibility promise of Windows in servicing all published Windows Console APIs for the indefinite future, while providing a replacement server-hosting interface that matched what is expected on all other platforms: virtual terminal sequences. As such, this effort performed the mirror image of the client phase: the *pseudoconsole* projects what would be displayed onto the screen as *virtual terminal sequences* for a delegated host and interprets replies into Windows-format input sequences for client application consumption.

Roadmap for the future

Terminal applications

[2019-Now] This is the open source era for the console subsystem, focusing on the new Windows Terminal. Announced during the Microsoft Build conference in May 2019, Windows Terminal is entirely on GitHub at microsoft/terminal. Building the Windows Terminal application on top of the refined platform for pseudoconsole will be the focus of this era, bringing a first-class terminal experience directly to developers on the Windows platform.

Windows Terminal intends not only to showcase the platform — including the WinUI interface technology, the MSIX packaging model, and the C++/WinRT component architecture — but also as a validation of the platform itself. Windows Terminal is driving the Windows organization to open and evolve the app platform as necessary to continue to lift the productivity of developers. The Windows Terminal unique set of power user and developer requirements drive the modern Windows platform requirements for what those markets truly need from Windows.

Inside the Windows operating system, this includes retiring the classic console host user interface from its default position in favor of Windows Terminal, ConPTY, and virtual terminal sequences.

Lastly, this era intends to offer full choice over the default experience, whether it is the Windows Terminal product or any alternative terminals.

Client support library

[Future] With the support and documentation of virtual terminal sequences on the client side, we strongly encourage Windows command-line utility developers to use virtual terminal sequences first over the classic Windows APIs to gain the benefit of a unified ecosystem with all platforms. However, one significant missing piece is that other platforms have a wide array of client-side helper libraries for handling input like readline and graphical display like ncurses. This particular future road map element represents the exploration of what the ecosystem offers and how we can accelerate the adoption of virtual terminal sequences in Windows command-line applications over the classic Console API.

Sequence Passthrough

[Future] The combination of virtual terminal client and server implementations allows the full mixing and matching of client command-line and terminal hosting applications. This combination can speak to either the classic Windows Console APIs or virtual terminal sequences, however, there is an overhead cost to translating this into the classic compatible Windows method and then back into the more universal virtual terminal method.

Once the market sufficiently adopts *virtual terminal sequences* and UTF-8 on Windows, the conversion/interpretation job of the console host can be optionally disabled. The console host would then become a simple API call servicer and relay from device calls to the hosting application via the pseudoconsole. This change will increase performance and maximize the dialect of sequences that can be spoken between the client application and the terminal. Through this change additional interactivity scenarios would be enabled and *(finally)* bring the Windows world into alignment with the family of all other platforms in the command-line application space.

Definitions

5/18/2021 • 4 minutes to read • Edit Online

This document provides the definitions of specific words and phrases in this space and be used as reference throughout this document set.

Command Line Applications

Command line applications, or sometimes called "console applications" and/or referred to as "clients" of the console subsystem, are programs that operate mainly on a stream of text or character information. They generally contain no user interface elements of their own and delegate both the output/display and the input/interaction roles to a hosting application. Command line applications receive a stream of text on their standard input `STDIN` handle which represents a user's keyboard input, process that information, then respond with a stream of text on their standard output `STDOUT` for display back to the user's monitor. Of course, this has evolved over time for additional input devices and remote scenarios, but the same basic philosophy remains the same: command-line clients operate on text and someone else manages display/input.

Standard Handles

The standard handles are a series, `STDIN`, `STDOUT`, and `STDERR`, introduced as part of a process space on startup. They represent a place for information to be accepted on the way in and sent back on the way out (including a special place to report errors out). For command-line applications, these must always exist when the application starts. They are either inherited from the parent automatically, set explicitly by the parent, or created automatically by the operating system if neither are specified/permitted. For classic Windows applications, these may be blank on startup. However, they can be implicitly or explicitly inherited from the parent or allocated, attached, and freed during runtime by the application itself.

Standard handles do not imply a specific type of attached device. In the case of command-line applications, however, the device is most commonly a console device, file (from redirection in a shell), or a pipe (from a shell connecting the output of one utility to the input of the next). It may also be a socket or any other type of device.

TTY/PTY

On non-Windows platforms, the TTY and PTY devices represent respectively either a true physical device or a software-created pseudo-device that are the same concept as a Windows console session: a channel where communication between a command-line client application and a server host interactivity application or physical keyboard/display device can exchange text-based information.

Clients and Servers

Within this space, we're referring to "clients" as applications that do the work of processing information and running commands. The "server" applications are those that are responsible for the user interface and are workers to translate input and output into standard forms on behalf of the clients.

Console Subsystem

This is a catch-all term representing all modules affecting console and command-line operations. It specifically refers to a flag that is a part of the Portable Executable header that specifies whether the starting application is either a command-line/console application (and must have standard handles to start) or a windows application (and does not need them).

The console host, command-line client applications, the console driver, the console API surface, the pseudoconsole infrastructure, terminals, configuration property sheets, the mechanisms and stubs inside the process loader, and any utilities related to the workings of these forms of applications are considered to belong to this group.

Console Host

The Windows Console Host, or `conhost.exe`, is both the server application for all of the Windows Console APIs as well as the classic Windows user interface for working with command-line applications. The complete contents of this binary, both the API server and the UI, historically belonged to Windows `csrss.exe`, a critical system process, and was diverged for security and isolation purposes. Going forward, `conhost.exe` will continue to be responsible for API call servicing and translation, but the user-interface components are intended to be delegated through a pseudoconsole to a terminal.

Pseudoconsole

This is the Windows simulation of a pseudoterminal or "PTY" from other platforms. It tries to match the general interface philosophy of PTYs, providing a simple bidirectional channel of text based communication, but it supplements it on Windows with a large compatibility layer to translate the breadth of Windows applications written prior to this design philosophy change from the classic console API surface into the simple text channel communication form. Terminals can use the pseudoconsole to take ownership of the user-interface elements away from the console host, `conhost.exe`, while leaving it in charge of the API servicing, translation, and compatibility efforts.

Terminal

A terminal is the user-interface and interaction module for a command-line application. Today, it's a software representation of what used to be historically a physical device with a display monitor, a keyboard, and a bidirectional serial communication channel. It is responsible for gathering input from the user in a variety of forms, translating it and encoding it and any special command information into a single text stream, and submitting it to the PTY for transmission on to the `STDIN` channel of the command-line client application. It is also responsible for receiving back information, via the PTY, that came from a client application's `STDOUT` channel, decoding any special information in the payload, laying out all the text and additional commands, and presenting that graphically to the end user.

Classic Console APIs versus Virtual Terminal Sequences

5/18/2021 • 8 minutes to read • Edit Online

Our recommendation is to replace the classic **Windows Console API** with **virtual terminal sequences**. This article will outline the difference between the two and discuss the reasons for our recommendation.

Definitions

The classic **Windows Console API** surface is defined as the series of C language functional interfaces on `kernel32.dll` with "Console" in the name.

Virtual terminal sequences is defined as a language of commands that's embedded in the standard input and standard output streams. Virtual terminal sequences use non-printable escape characters for signaling commands interleaved with normal printable text.

History

The **Windows Console** provides a broad API surface for client command-line applications to manipulate both the output display buffer and the user input buffer. However, other non-Windows platforms have never afforded this specific API-driven approach to their command-line environments, choosing instead to use virtual terminal sequences embedded within the standard input and standard output streams. *(For a time, Microsoft supported this behavior too in early editions of DOS and Windows through a driver called ANSI.SYS.)*

By contrast, **virtual terminal sequences** (in a variety of dialects) drive the command-line environment operations for all other platforms. These sequences are rooted in an ECMA Standard and series of extensions by many vendors tracing back to Digital Equipment Corporation and Tektronix terminals, through to more modern and common software terminals, like xterm. Many extensions exist within the virtual terminal sequence domain and some sequences are more widely supported than others, but it is safe to say that the world has standardized on this as the command language for command-line experiences with a well-known subset being supported by virtually every terminal and command-line client application.

Cross-Platform Support

Virtual terminal sequences are natively supported across platforms, making terminal applications and command-line utilities easily portable between versions and variations of operating systems, with the exception of Windows.

By contrast, **Windows Console APIs** are only supported on Windows. An extensive adapter or translation library must be written between Windows and virtual terminal, or vice-versa, when attempting to port command-line utilities from one platform or another.

Remote Access

Virtual terminal sequences hold a major advantage for remote access. They require no additional work to transport, or perform remote procedure calls, over what is required to set up a standard remote command-line connection. Simply connecting an outbound and an inbound transport channel (or a single bidirectional channel) over a pipe, socket, file, serial port, or any other device is sufficient to completely carry all information required for an application speaking these sequences to a remote host.

On the contrary, the **Windows Console APIs** have only been accessible on the local machine and all efforts to

remote them would require building an entire remote calling and transport interface layer beyond just a simple channel.

Separation of Concerns

Some **Windows Console APIs** provide low-level access to the input and output buffers or convenience functions for interactive command-lines. This might include aliases and command history programmed within the console subsystem and host environment, instead of within the command-line client application itself.

By contrast, **other platforms** make memory of the current state of the application and convenience functionality the responsibility of the command-line utility or shell itself.

The **Windows Console** way of handling this responsibility in the console host and API makes it quicker and easier to write a command-line application with these features, removing the responsibility of remembering drawing state or handling editing convenience features. However, this makes it nearly impossible to connect those activities remotely across platforms, versions, or scenarios due to variations in implementations and availability. This way of handling responsibility also makes the final interactive experience of these Windows command-line applications completely dependent on the console host's implementation, priorities, and release cycle.

For example, advanced line editing features, like syntax highlighting and complex selection, are only possible when a command-line application handles editing concerns itself. The console could never have enough context to fully understand these scenarios in a broad manner like the client application can.

By contrast, other platforms use **virtual terminal sequences** to handle these activities and virtual terminal communication itself through reusable client-side libraries, like readline and ncurses. The final terminal is only responsible for displaying information and receiving input through that bidirectional communication channel.

Wrong-Way Verbs

With **Windows Console**, some actions can be performed in the opposite-to-natural direction on the input and output streams. This allows Windows command-line applications to avoid the concern of managing their own buffers. It also allows Windows command-line apps to perform advanced operations, like simulating/injecting input on behalf of a user, or reading back some of the history of what was written.

While this provides additional power to Windows applications operating in a specific user-context on a single machine, it also provides a vector to cross security and privilege-levels or domains when used in certain scenarios. Such scenarios include operating between contexts on the same machine, or across contexts to another machine or environment.

Other platforms, which use **virtual terminal sequences**, do not allow this activity. The intent of our recommendation to transition from classic Windows Console to virtual terminal sequences is to converge with this strategy for both interoperability and security reasons.

Direct Window Access

Windows Console API surface provides the exact window handle to the hosting window. This allows a command-line utility to perform advanced window operations by reaching into the wide gamut of Win32 APIs permitted against a window handle. These Win32 APIs can manipulate the window state, frame, icon, or other properties about the window.

By contrast, on other platforms with **virtual terminal sequences**, there is a narrow set of commands that can be performed against the window. These commands can do things like changing the window size or displayed title, but they must be done in the same band and under the same control as the remainder of the stream.

As Windows has evolved, the security controls and restrictions on window handles have increased. Additionally, the nature and existence of an application-addressable window handle on any specific user interface element has evolved, especially with the increased support of device form factors and platforms. This makes direct window access to command-line applications fragile as the platform and experiences evolve.

Unicode

UTF-8 is the accepted encoding for Unicode data across almost all modern platforms, as it strikes the right balance between portability, storage size and processing time. However, Windows historically chose UTF-16 as its primary encoding for Unicode data. Support for UTF-8 is increasing in Windows and use of these Unicode formats does not preclude the usage of other encodings.

The **Windows Console** platform has supported and will continue to support all existing code pages and encodings. Use UTF-16 for maximum compatibility across Windows versions and perform algorithmic translation with UTF-8 if necessary. Increased support of UTF-8 is in progress for the console system.

UTF-16 support in the console can be utilized with no additional configuration via the *W* variant of all console APIs and is a more likely choice for applications already well versed in UTF-16 through communication with the `wchar_t` and *W* variant of other Microsoft and Windows platform functions and products.

UTF-8 support in the console can be utilized via the *A* variant of Console APIs against console handles after setting the codepage to `65001` or `CP_UTF8` with the **SetConsoleOutputCP** and **SetConsoleCP** methods, as appropriate. Setting the code pages in advance is only necessary if the machine has not chosen "Use Unicode UTF-8 for worldwide language support" in the settings for Non-Unicode applications in the Region section of the Control Panel.

> **NOTE**
> As of now, UTF-8 is supported fully on the standard output stream with the **WriteConsole** and **WriteFile** methods. Support on the input stream varies depending on the input mode and will continue to improve over time. Notably the default **"cooked"** modes on input do not fully support UTF-8 yet. The current status of this work can be found at **microsoft/terminal#7777** on GitHub. The workaround is to use the algorithmically-translatable UTF-16 for reading input through **ReadConsoleW** or **ReadConsoleInputW** until the outstanding issues are resolved.

Recommendations

For all new and ongoing development on Windows, **virtual terminal sequences are recommended** as the way of interacting with the terminal. This will converge Windows command-line client applications with the style of application programming on all other platforms.

Exceptions for using Windows Console APIs

A **limited subset of Windows Console APIs is still necessary** to establish the initial environment. The Windows platform still differs from others in process, signal, device, and encoding handling:

- The standard handles to a process will still be controlled with **GetStdHandle** and **SetStdHandle**.
- Configuration of the console modes on a handle to opt in to Virtual Terminal Sequence support will be handled with **GetConsoleMode** and **SetConsoleMode**.
- Declaration of code page or UTF-8 support is conducted with **SetConsoleOutputCP** and **SetConsoleCP** methods.
- Some level of overall process management may be required with the **AllocConsole**, **AttachConsole** and **FreeConsole** to join or leave a console device session.
- Signals and signal handling will continue to be conducted with **SetConsoleCtrlHandler**, **HandlerRoutine**, and **GenerateConsoleCtrlEvent**.
- Communication with the console device handles can be conducted with **WriteConsole** and **ReadConsole**. These may also be leveraged through programming language runtimes in the forms of: - C Runtime (CRT): Stream I/O like **printf**, **scanf**, **putc**, **getc**, or other levels of I/O functions. - C++ Standard Library (STL): iostream like **cout** and **cin**. - .NET Runtime: System.Console like

Console.WriteLine.

- Applications that must be aware of window size changes will still need to use **ReadConsoleInput** to receive them interleaved with key events as **ReadConsole** alone will discard them.
- Finding the window size must still be performed with **GetConsoleScreenBufferInfo** for applications attempting to draw columns, grids, or fill the display. Window and buffer size will match in a pseudoconsole session.

Future planning and pseudoconsole

There are no plans to remove the Windows console APIs from the platform.

On the contrary, the Windows Console host has provided the **pseudoconsole** technology to translate existing Windows command-line application calls into virtual terminal sequences and forward them to another hosting environment remotely or across platforms.

This translation is not perfect. It requires the console host window to maintain a simulated environment of what Windows would have displayed to the user. It then projects a replica of this simulated environment to the **pseudoconsole** host. All Windows Console API calls are operated within the simulated environment to serve the needs of the legacy command-line client application. Only the effects are propagated onto the final host.

A command-line application desiring full compatibility across platforms and full support of all new features and scenarios both on Windows and elsewhere is therefore recommended to move to virtual terminal sequences and adjust the architecture of command-line applications to align with all platforms.

More information about this Windows transition for command-line applications can be found on our ecosystem roadmap.

About Character Mode Applications

5/18/2021 • 2 minutes to read • Edit Online

Character mode (or "command-line") applications:

1. [Optionally] Read data from standard input (stdin)
2. Do "work"
3. [Optionally] Write data to standard output (stdout) or standard error (stderr)

Character mode applications communicate with the end-user through a "console" (or "terminal") application. A console converts user input from keyboard, mouse, touch-screen, pen, etc., and sends it to a character mode application's stdin. A console may also display a character mode application's text output on the user's screen.

In Windows, the console is built-in and provides a rich API through which character mode applications can interact with the user. However, in the recent era, the console team is encouraging all character mode applications to be developed with virtual terminal sequences over the classic API calls for maximum compatibility between Windows and other operating systems. More details on this transition and the trade offs involved can be found in our discussion of classic APIs versus virtual terminal sequences.

- Consoles
- Input and Output Methods
- Console Code Pages
- Console Control Handlers
- Console Aliases
- Console Buffer Security and Access Rights
- Console Application Issues

Input and Output Methods

9/16/2021 • 2 minutes to read • Edit Online

There are two different approaches to console I/O, the choice of which depends on how much flexibility and control an application needs. The high-level approach enables simple character stream I/O, but it limits access to a console's input and screen buffers. The low-level approach requires that developers write more code and choose among a greater range of functions, but it also gives an application more flexibility.

> **NOTE**
> The low-level approach is not recommended for new and ongoing development. Applications needing functionality from the low-level console I/O functions are encouraged to use **virtual terminal sequences** and explore our documentation on both **classic functions versus virtual terminal** and **the ecosystem roadmap**.

An application can use the file I/O functions, **ReadFile** and **WriteFile**, and the console functions, **ReadConsole** and **WriteConsole**, for high-level I/O that provides indirect access to a console's input and screen buffers. The high-level input functions filter and process the data in a console's input buffer to return input as a stream of characters, discarding mouse and buffer-resizing input. Similarly, the high-level output functions write a stream of characters that are displayed at the current cursor location in a screen buffer. An application controls the way these functions work by setting a console's I/O modes.

The low-level I/O functions provide direct access to a console's input and screen buffers, enabling an application to access mouse and buffer-resizing input events and extended information for keyboard events. Low-level output functions enable an application to read from or write to a specified number of consecutive character cells in a screen buffer, or to read or write to rectangular blocks of character cells at a specified location in a screen buffer. A console's input modes affect low-level input by enabling the application to determine whether mouse and buffer-resizing events are placed in the input buffer. A console's output modes have no effect on low-level output.

The high-level and low-level I/O methods are not mutually exclusive, and an application can use any combination of these functions. Typically, however, an application uses one approach or the other exclusively and we recommend focusing on one particular paradigm for optimal results.

> **TIP**
> The ideal forward looking application will focus on the high-level methods and augment further needs with **virtual terminal sequences** through the high-level I/O methods when necessary avoiding the use of low-level I/O functions entirely.

The following topics describe the console modes and the high-level and low-level I/O functions.

- Console Modes
- High-Level Console I/O
- High-Level Console Modes
- High-Level Console Input and Output Functions
- Console Virtual Terminal Sequences
- Classic Functions versus Virtual Terminal Sequences
- Ecosystem Roadmap
- Low-Level Console I/O

- Low-Level Console Modes
- Low-Level Console Input Functions
- Low-Level Console Output Functions

Console Code Pages

5/18/2021 • 2 minutes to read • Edit Online

A *code page* is a mapping of 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages.

Associated with each console are two code pages: one for input and one for output. A console uses its input code page to translate keyboard input into the corresponding character value. It uses its output code page to translate the character values written by the various output functions into the images displayed in the console window. An application can use the **SetConsoleCP** and **GetConsoleCP** functions to set and retrieve a console's input code pages and the **SetConsoleOutputCP** and **GetConsoleOutputCP** functions to set and retrieve its output code pages.

The identifiers of the code pages available on the local computer are stored in the registry under the following key: `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage`

For information about using the registry functions to determine the available code pages, see **Registry**.

> **TIP**
>
> It is recommended for all new and updated command-line applications to avoid code pages and use **Unicode**. UTF-16 formatted text can be sent to the *W* family of console APIs. UTF-8 formatted text can be sent to the *A* family of console APIs after ensuring the code page is first set to **65001 (CP_UTF8)** with the **SetConsoleCP** and **SetConsoleOutputCP** functions.

Console Control Handlers

9/16/2021 • 2 minutes to read • Edit Online

Each console process has its own list of control handler functions that are called by the system when the process receives a CTRL+C, CTRL+BREAK, or CTRL+CLOSE signal. Initially, the list of control handlers for each process contains only a default handler function that calls the **ExitProcess** function. A console process can add or remove additional **HandlerRoutine** functions by calling the **SetConsoleCtrlHandler** function. This function does not affect the lists of control handlers for other processes. When a console process receives any of the control signals, it calls the handler functions on a last-registered, first-called basis until one of the handlers returns **TRUE**. If none of the handlers returns **TRUE**, the default handler is called.

The function's *dwCtrlType* parameter identifies which control signal was received, and the return value indicates whether the signal was handled.

A new thread is started inside the command-line client process to run the handler routines. More information on the timeout values and action of this thread can be found in the **HandlerRoutine** function documentation.

For an example of a control handler function, see Registering a Control Handler Function.

Note that calling **AttachConsole**, **AllocConsole**, or **FreeConsole** will reset the table of control handlers in the client process to its initial state.

Console Aliases

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Console aliases are used to map source strings to target strings. For example, you can define a console alias that maps "test" to "cd \a_very_long_path\test". When you type "test" at the command line, the console subsystem expands the alias and executes the specified cd command.

To define a console alias, use **Doskey.exe** to create a macro, or use the **AddConsoleAlias** function. The following example uses `Doskey.exe`:

doskey test=cd *a_very_long_path*\test

The following call to **AddConsoleAlias** creates the same console alias:

```
AddConsoleAlias( TEXT("test"),
                 TEXT("cd \\<a_very_long_path>\\test"),
                 TEXT("cmd.exe"));
```

To add parameters to a console alias macro using `Doskey.exe`, use the batch parameters `$1` through `$9`. For more information on the special codes that can be used in Doskey macro definitions, see the command-line help for `Doskey.exe` or Doskey on TechNet.

All instances of an executable file running in the same console window share any defined console aliases. Multiple instances of the same executable file running in different console windows do not share console aliases. Different executable files running in the same console window do not share console aliases.

To retrieve the target string for a specified source string and executable file, use the **GetConsoleAlias** function. To retrieve all aliases for a specified executable file, use the **GetConsoleAliases** function. To retrieve the names of all aliases for which console aliases have been defined, use the **GetConsoleAliasExes** function.

Console Buffer Security and Access Rights

5/18/2021 • 2 minutes to read • Edit Online

The Windows security model enables you to control access to console input buffers and console screen buffers. For more information about security, see Access-Control Model.

Console Object Security Descriptors

You can specify a security descriptor for the console input and console screen buffers when you call the **CreateFile** or **CreateConsoleScreenBuffer** function. If you specify **NULL**, the object gets a default security descriptor. The ACLs in the default security descriptor for a console buffer come from the primary or impersonation token of the creator.

The handles returned by **CreateFile**, **CreateConsoleScreenBuffer**, and **GetStdHandle** have the **GENERIC_READ** and **GENERIC_WRITE** access rights.

The valid access rights include the **GENERIC_READ** and **GENERIC_WRITE** generic access rights.

VALUE	MEANING
GENERIC_READ (0x80000000L)	Requests read access to the console screen buffer, enabling the process to read data from the buffer.
GENERIC_WRITE (0x40000000L)	Requests write access to the console screen buffer, enabling the process to write data to the buffer.

NOTE

Universal Windows Platform console apps and those with a lower **integrity level** than the attached console will be prohibited from both reading the output buffer and writing to the input buffer even if the security descriptors above would normally permit it. Please see the **Wrong Way Verbs** discussion below for more details.

Wrong-Way Verbs

Some operations to the console objects will be denied even if the object has a security descriptor that is stated to specifically permit reading or writing. This specifically concerns command-line applications running in a reduced-privilege context that are sharing a console session that was created by a command-line application in a more permissive context.

The term "wrong-way verbs" is intended to apply to the operation that is the converse of the normal flow for one of the console objects. Specifically, the normal flow for the output buffer is writing and the normal flow for the input buffer is reading. The "wrong-way" would therefore be the reading of the output buffer or the writing of the input buffer. These are functions that are described in the **Low-Level Console I/O Functions** documentation.

The two scenarios where this can be found are:

1. **Universal Windows Platform console apps**. As these are cousins of other Universal Windows Platform applications, they hold a promise that they are isolated from other applications and provide user guarantees around the effects of their operation.
2. Any console application intentionally launched with a lower **integrity level** than the existing session which

can be accomplished with **labeling or token manipulation during CreateProcess**.

If either of these scenarios is detected, the console will apply the "wrong-way verbs" flag to the command-line application connection and reject calls to the following APIs to reduce the surface of communication between the levels:

ReadConsoleOutput
ReadConsoleOutputCharacter
ReadConsoleOutputAttribute

WriteConsoleInput

Rejected calls will receive an **access denied** error code, the same as if the read or write permission were denied by the security descriptors on the object.

Console Application Issues

5/18/2021 • 2 minutes to read • Edit Online

The 8-bit console functions use the OEM code page. All other functions use the ANSI code page by default. This means that strings returned by the console functions may not be processed correctly by the other functions and vice versa. For example, if **FindFirstFileA** returns a string that contains certain extended ANSI characters, **WriteConsoleA** will not display the string properly.

The best long-term solution for a console application is to use **Unicode**. The console will accept UTF-16 encoding on the W variant of the APIs or UTF-8 encoding on the A variant of the APIs after using **SetConsoleCP** and **SetConsoleOutputCP** to `65001` (`CP_UTF8` constant) for the UTF-8 code page.

Barring that solution, a console application should use the SetFileApisToOEM function. That function changes relevant file functions so that they produce OEM character set strings rather than ANSI character set strings.

The following are file functions:

CopyFile
CreateDirectory
CreateFile
CreateProcess
DeleteFile
FindFirstFile
FindNextFile
GetCurrentDirectory
GetDiskFreeSpace
GetDriveType

GetFileAttributes
GetFullPathName
GetModuleFileName
GetModuleHandle
GetSystemDirectory
GetTempFileName
GetTempPath
GetVolumeInformation
GetWindowsDirectory
LoadLibrary

LoadLibraryEx
MoveFile
MoveFileEx
OpenFile
RemoveDirectory
SearchPath
SetCurrentDirectory
SetFileAttributes

When dealing with command lines, a console application should obtain the command line in Unicode form and convert it to OEM form, using the relevant character-to-OEM functions. Note, also, that *argv* uses the ANSI character set.

Consoles

5/18/2021 • 2 minutes to read • Edit Online

A *console* is an application that provides I/O services to character-mode applications.

A console consists of an input buffer and one or more screen buffers. The *input buffer* contains a queue of input records, each of which contains information about an input event. The input queue always includes key-press and key-release events. It may also include mouse events (pointer movements and button presses and releases) and events during which user actions affect the size of the active screen buffer. A *screen buffer* is a two-dimensional array of character and color data for output in a console window. Any number of processes can share a console.

> **TIP**
> A broader idea of consoles and how they relate to terminals and command-line client applications can be found in the **ecosystem roadmap**.

- Creation of a Console
- Attaching to a Console
- Closing a Console
- Console Handles
- Console Input Buffer
- Console Screen Buffers
- Window and Screen Buffer Size
- Console Selection
- Scrolling the Screen Buffer

Creation of a Console

9/16/2021 • 3 minutes to read • Edit Online

The system creates a new console when it starts a *console process*, a character-mode process whose entry point is the **main** function. For example, the system creates a new console when it starts the command processor `cmd.exe`. When the command processor starts a new console process, the user can specify whether the system creates a new console for the new process or whether it inherits the command processor's console.

A process can create a console by using one of the following methods:

- A graphical user interface (GUI) or console process can use the **CreateProcess** function with **CREATE_NEW_CONSOLE** to create a console process with a new console. (By default, a console process inherits its parent's console, and there is no guarantee that input is received by the process for which it was intended.)
- A GUI or console process that is not currently attached to a console can use the **AllocConsole** function to create a new console. (GUI processes are not attached to a console when they are created. Console processes are not attached to a console if they are created using **CreateProcess** with **DETACHED_PROCESS**.)

Typically, a process uses **AllocConsole** to create a console when an error occurs requiring interaction with the user. For example, a GUI process can create a console when an error occurs that prevents it from using its normal graphical interface, or a console process that does not normally interact with the user can create a console to display an error.

A process can also create a console by specifying the **CREATE_NEW_CONSOLE** flag in a call to **CreateProcess**. This method creates a new console that is accessible to the child process but not to the parent process. Separate consoles enable both parent and child processes to interact with the user without conflict. If this flag is not specified when a console process is created, both processes are attached to the same console, and there is no guarantee that the correct process will receive the input intended for it. Applications can prevent confusion by creating child processes that do not inherit handles of the input buffer, or by enabling only one child process at a time to inherit an input buffer handle while preventing the parent process from reading console input until the child has finished.

Creating a new console results in a new console window, as well as separate I/O buffers for output to the screen and input from the user. The process associated with the new console uses the **GetStdHandle** function to get the handles of the new console's input and screen buffers. These handles enable the process to access the console.

When a process uses **CreateProcess**, it can specify a **STARTUPINFO** structure, whose members control the characteristics of the first new console (if any) created for the child process. The **STARTUPINFO** structure specified in the call to **CreateProcess** affects a console created if the **CREATE_NEW_CONSOLE** flag is specified. It also affects a console created if the child process subsequently uses **AllocConsole**. The following console characteristics can be specified:

- Size of the new console window, in character cells
- Location of the new console window, in screen pixel coordinates
- Size of the new console's screen buffer, in character cells
- Text and background color attributes of the new console's screen buffer
- Display name for the title bar of the new console's window

The system uses default values if the **STARTUPINFO** values are not specified. A child process can use the **GetStartupInfo** function to determine the values in its **STARTUPINFO** structure.

A process cannot change the location of its console window on the screen, but the following console functions are available to set or retrieve the other properties specified in the STARTUPINFO structure.

FUNCTION	DESCRIPTION
GetConsoleScreenBufferInfo	Retrieves the window size, screen buffer size, and color attributes.
SetConsoleWindowInfo	Changes the size of the console window.
SetConsoleScreenBufferSize	Changes the size of the console screen buffer.
SetConsoleTextAttribute	Sets the color attributes.
SetConsoleTitle	Sets the console window title.
GetConsoleTitle	Retrieves the console window title.

A process can use the FreeConsole function to detach itself from an inherited console or from a console created by AllocConsole.

A process can use the AttachConsole function to attach itself to another existing console session after using FreeConsole to detach from its own session (or if there is otherwise no attached session).

Attaching to a Console

5/18/2021 • 2 minutes to read • Edit Online

A process can use the **AttachConsole** function to attach to a console as a client. A process can be attached to one console.

A console can have many processes attached to it. To retrieve a list of the processes attached to a console, call the **GetConsoleProcessList** function.

To attach as a server, see information about **Pseudoconsoles**.

Closing a Console

5/18/2021 • 2 minutes to read • Edit Online

A process can use the **FreeConsole** function to detach itself from its console. If other processes share the console, the console is not destroyed, but the process that called **FreeConsole** cannot refer to it. After calling **FreeConsole**, the process can use **AllocConsole** to create a new console or **AttachConsole** to attach to another console.

A console is closed when the last process attached to it terminates or calls **FreeConsole**.

Console Handles

5/18/2021 • 3 minutes to read • Edit Online

A console process uses handles to access the input and screen buffers of its console. A process can use the **GetStdHandle**, **CreateFile**, or **CreateConsoleScreenBuffer** function to open one of these handles.

The **GetStdHandle** function provides a mechanism for retrieving the standard input (`STDIN`), standard output (`STDOUT`), and standard error (`STDERR`) handles associated with a process. During console creation, the system creates these handles. Initially, `STDIN` is a handle to the console's input buffer, and `STDOUT` and `STDERR` are handles of the console's active screen buffer. However, the **SetStdHandle** function can redirect the standard handles by changing the handle associated with `STDIN` , `STDOUT` , or `STDERR` . Because the parent's standard handles are inherited by any child process, subsequent calls to **GetStdHandle** return the redirected handle. A handle returned by **GetStdHandle** may, therefore, refer to something other than console I/O. For example, before creating a child process, a parent process can use **SetStdHandle** to set a pipe handle to be the `STDIN` handle that is inherited by the child process. When the child process calls **GetStdHandle**, it gets the pipe handle. This means that the parent process can control the standard handles of the child process. The handles returned by **GetStdHandle** have `GENERIC_READ | GENERIC_WRITE` access unless **SetStdHandle** has been used to set the standard handle to have lesser access.

The value of the handles returned by **GetStdHandle** are not 0, 1, and 2, so the standard predefined stream constants in Stdio.h (`STDIN` , `STDOUT` , and `STDERR`) cannot be used in functions that require a console handle.

The **CreateFile** function enables a process to get a handle to its console's input buffer and active screen buffer, even if `STDIN` and `STDOUT` have been redirected. To open a handle to a console's input buffer, specify the `CONIN$` value in a call to **CreateFile**. Specify the `CONOUT$` value in a call to **CreateFile** to open a handle to a console's active screen buffer. **CreateFile** enables you to specify the read/write access of the handle that it returns.

The **CreateConsoleScreenBuffer** function creates a new screen buffer and returns a handle. This handle can be used in any function that accepts a handle to console output. The new screen buffer is not active (displayed) until its handle is specified in a call to the **SetConsoleActiveScreenBuffer** function. Note that changing the active screen buffer does not affect the handle returned by **GetStdHandle**. Similarly, using **SetStdHandle** to change the `STDOUT` handle does not affect the active screen buffer.

Console handles returned by **CreateFile** and **CreateConsoleScreenBuffer** can be used in any of the console functions that require a handle to a console's input buffer or of a console screen buffer. Handles returned by **GetStdHandle** can be used by the console functions if they have not been redirected to refer to something other than console I/O. If a standard handle has been redirected to refer to a file or a pipe, however, the handle can only be used by the **ReadFile** and **WriteFile** functions. **GetFileType** can assist in determining what device type the handle refers to. A console handle presents as `FILE_TYPE_CHAR` .

A process can use the **DuplicateHandle** function to create a duplicate console handle that has different access or inheritability from the original handle. Note, however, that a process can create a duplicate console handle only for its own use. This differs from other handle types (such as file, pipe, or mutex objects), for which **DuplicateHandle** can create a duplicate that is valid for a different process. Access to a console must be shared during creation of the other process or may be requested by the other process through the **AttachConsole** mechanism.

To close a console handle, a process can use the **CloseHandle** function.

Console Input Buffer

5/18/2021 • 4 minutes to read • Edit Online

Each console has an input buffer that contains a queue of input event records. When a console's window has the keyboard focus, a console formats each input event (such as a single keystroke, a movement of the mouse, or a mouse-button click) as an input record that it places in the console's input buffer.

Applications can access a console's input buffer indirectly by using the high-level console I/O functions, or directly by using the low-level console input functions. The high-level input functions filter and process the data in the input buffer, returning only a stream of input characters. The low-level input functions enable applications to read input records directly from a console's input buffer, or to place input records into the input buffer. To open a handle to a console's input buffer, specify the **CONIN$** value in a call to the **CreateFile** function.

An input record is a structure containing information about the type of event that occurred (keyboard, mouse, window resizing, focus, or menu event) as well as specific details about the event. The **EventType** member in an **INPUT_RECORD** structure indicates which type of event is contained in the record.

Focus and menu events are placed in a console's input buffer for internal use by the system and should be ignored by applications.

Keyboard Events

Keyboard events are generated when any key is pressed or released; this includes control keys. However, the ALT key has special meaning to the system when pressed and released without being combined with another character, and it is not passed through to the application. Also, the CTRL+C key combination is not passed through if the input handle is in processed mode.

If the input event is a keystroke, the **Event** member in **INPUT_RECORD** is a **KEY_EVENT_RECORD** structure containing the following information:

- A Boolean value indicating whether the key was pressed or released.
- A repeat count that can be greater than one when a key is held down.
- The virtual-key code, identifying the given key in a device-independent manner.
- The virtual-scan code, indicating the device-dependent value generated by the keyboard hardware.
- The translated Unicode™ or ANSI character.
- A flag variable indicating the state of the control keys (the ALT, CTRL, SHIFT, NUM LOCK, SCROLL LOCK, and CAPS LOCK keys) and indicating whether an enhanced key was pressed. Enhanced keys for the IBM® 101-key and 102-key keyboards are the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad and the divide (/) and ENTER keys in the numeric keypad.

Mouse Events

Mouse events are generated whenever the user moves the mouse or presses or releases one of the mouse buttons. Mouse events are placed in the input buffer only if the following conditions are met:

- The console input mode is set to **ENABLE_MOUSE_INPUT** (the default mode).
- The console window has the keyboard focus.
- The mouse pointer is within the borders of the console's window.

If the input event is a mouse event, the **Event** member in **INPUT_RECORD** is a **MOUSE_EVENT_RECORD** structure containing the following information:

- The coordinates of the mouse pointer in terms of the character-cell row and column in the console screen buffer's coordinate system.
- A flag variable indicating the state of the mouse buttons.
- A flag variable indicating the state of the control keys (ALT, CTRL, SHIFT, NUM LOCK, SCROLL LOCK, and CAPS LOCK) and indicating whether an enhanced key was pressed. Enhanced keys for the IBM 101-key and 102-key keyboards are the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad and the divide (/) and ENTER keys in the numeric keypad.
- A flag variable indicating whether the event was a normal button-press or button-release event, a mouse movement event, or the second click of a double-click event.

> **NOTE**
>
> The mouse position coordinates are in terms of the console screen buffer, not the console window. The screen buffer may have been scrolled with respect to the window, so the upper left corner of the window is not necessarily the (0,0) coordinate of the console screen buffer. To determine the coordinates of the mouse relative to the coordinate system of the window, subtract the window origin coordinates from the mouse position coordinates. Use the **GetConsoleScreenBufferInfo** function to determine the window origin coordinates.

The **dwButtonState** member of the **MOUSE_EVENT_RECORD** structure has a bit corresponding to each mouse button. The bit is 1 if the button is down and 0 if the button is up. A button-release event is detected by a 0 value for the **dwEventFlags** member of **MOUSE_EVENT_RECORD** and a change in a button's bit from 1 to 0. The **GetNumberOfConsoleMouseButtons** function retrieves the number of buttons on the mouse.

Buffer-Resizing Events

A console window's menu enables the user to change the size of the active screen buffer; this change generates a buffer-resizing event. Buffer-resizing events are placed in the input buffer if the console's input mode is set to **ENABLE_WINDOW_INPUT** (that is, the default mode is disabled).

If the input event is a buffer-resizing event, the **Event** member of **INPUT_RECORD** is a **WINDOW_BUFFER_SIZE_RECORD** structure containing the new size of the console screen buffer, expressed in character-cell columns and rows.

If the user reduces the size of the console screen buffer, any data in the discarded portion of the buffer is lost.

Changes to the console screen buffer size as a result of application calls to the **SetConsoleScreenBufferSize** function are not generated as buffer-resizing events.

Console Screen Buffers

9/16/2021 • 6 minutes to read • Edit Online

A *screen buffer* is a two-dimensional array of character and color data for output in a console window. A console can have multiple screen buffers. The *active screen buffer* is the one that is displayed on the screen.

The system creates a screen buffer whenever it creates a new console. To open a handle to a console's active screen buffer, specify the **CONOUT$** value in a call to the **CreateFile** function. A process can use the **CreateConsoleScreenBuffer** function to create additional screen buffers for its console. A new screen buffer is not active until its handle is specified in a call to the **SetConsoleActiveScreenBuffer** function. However, screen buffers can be accessed for reading and writing whether they are active or inactive.

Each screen buffer has its own two-dimensional array of character information records. The data for each character is stored in a **CHAR_INFO** structure that specifies the Unicode or ANSI character and the foreground and background colors in which that character is displayed.

A number of properties associated with a screen buffer can be set independently for each screen buffer. This means that changing the active screen buffer can have a dramatic effect on the appearance of the console window. The properties associated with a screen buffer include:

- Screen buffer size, in character rows and columns.
- Text attributes (foreground and background colors for displaying text to be written by the **WriteFile** or **WriteConsole** function).
- Window size and location (the rectangular region of the console screen buffer that is displayed in the console window).
- Cursor position, appearance, and visibility.
- Output modes (**ENABLE_PROCESSED_OUTPUT** and **ENABLE_WRAP_AT_EOL_OUTPUT**). For more information about console output modes, see High-Level Console Modes.

When a screen buffer is created, it contains space characters in every position. Its cursor is visible and positioned at the buffer's origin (0,0), and the window is positioned with its upper left corner at the buffer's origin. The size of the console screen buffer, the window size, the text attributes, and the appearance of the cursor are determined by the user or by the system defaults. To retrieve the current values of the various properties associated with the console screen buffer, use the **GetConsoleScreenBufferInfo**, **GetConsoleCursorInfo**, and **GetConsoleMode** functions.

Applications that change any of the console screen buffer properties should either create their own screen buffer or save the state of the inherited screen buffer during startup and restore it at exit. This cooperative behavior is required to ensure that other applications sharing the same console session are not impacted by the changes.

> **TIP**
> It is recommended to use the **alternate buffer mode** going forward, if possible, instead of creating a second screen buffer for this purpose. **Alternate buffer mode** offers increased compatibility across remote devices and with other platforms. Please see our discussion on **classic console APIs versus virtual terminal** for more information.

Cursor Appearance and Position

A screen buffer's cursor can be visible or hidden. When it is visible, its appearance can vary, ranging from

completely filling a character cell to appearing as a horizontal line at the bottom of the cell. To retrieve information about the appearance and visibility of the cursor, use the **GetConsoleCursorInfo** function. This function reports whether the cursor is visible and describes the appearance of the cursor as the percentage of a character cell that it fills. To set the appearance and visibility of the cursor, use the **SetConsoleCursorInfo** function.

Characters written by the high-level console I/O functions are written at the current cursor location, advancing the cursor to the next location. To determine the current cursor position in the coordinate system of a screen buffer, use **GetConsoleScreenBufferInfo**. You can use **SetConsoleCursorPosition** to set the cursor position and, thereby, control the placement of text that is written or echoed by the high-level I/O functions. If you move the cursor, text at the new cursor location is overwritten.

> **NOTE**
> Using the low-level functions to find the cursor position is discouraged. It is recommended to use virtual terminal sequences to query this position if necessary for advanced layouts. More information about preferring virtual terminal sequences can be found in the **classic functions versus virtual terminal** document.

The position, appearance, and visibility of the cursor are set independently for each screen buffer.

Character Attributes

Character attributes can be divided into two classes: color and DBCS. The following attributes are defined in the `WinCon.h` header file.

ATTRIBUTE	MEANING
FOREGROUND_BLUE	Text color contains blue.
FOREGROUND_GREEN	Text color contains green.
FOREGROUND_RED	Text color contains red.
FOREGROUND_INTENSITY	Text color is intensified.
BACKGROUND_BLUE	Background color contains blue.
BACKGROUND_GREEN	Background color contains green.
BACKGROUND_RED	Background color contains red.
BACKGROUND_INTENSITY	Background color is intensified.
COMMON_LVB_LEADING_BYTE	Leading byte.
COMMON_LVB_TRAILING_BYTE	Trailing byte.
COMMON_LVB_GRID_HORIZONTAL	Top horizontal.
COMMON_LVB_GRID_LVERTICAL	Left vertical.
COMMON_LVB_GRID_RVERTICAL	Right vertical.

ATTRIBUTE	MEANING
COMMON_LVB_REVERSE_VIDEO	Reverse foreground and background attributes.
COMMON_LVB_UNDERSCORE	Underscore.

The foreground attributes specify the text color. The background attributes specify the color used to fill the cell's background. The other attributes are used with DBCS.

An application can combine the foreground and background constants to achieve different colors. For example, the following combination results in bright cyan text on a blue background.

```
FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_INTENSITY | BACKGROUND_BLUE
```

If no background constant is specified, the background is black, and if no foreground constant is specified, the text is black. For example, the following combination produces black text on a white background. Red, green, and blue are specified for the background which combines to a white background. No flag colors are specified for the foreground so it is black.

```
BACKGROUND_BLUE | BACKGROUND_GREEN | BACKGROUND_RED
```

Each screen buffer character cell stores the color attributes for the colors used in drawing the foreground (text) and background of that cell. An application can set the color data for each character cell individually, storing the data in the **Attributes** member of the CHAR_INFO structure for each cell. The current text attributes of each screen buffer are used for characters subsequently written or echoed by the high-level functions.

An application can use **GetConsoleScreenBufferInfo** to determine the current text attributes of a screen buffer and the **SetConsoleTextAttribute** function to set the character attributes. Changing a screen buffer's attributes does not affect the display of characters previously written. These text attributes do not affect characters written by the low-level console I/O functions (such as the **WriteConsoleOutput** or **WriteConsoleOutputCharacter** function), which either explicitly specify the attributes for each cell that is written or leave the attributes unchanged.

> **NOTE**
> Using the low-level functions to manipulate default and specific text attributes is discouraged. It is recommended to use virtual terminal sequences to set text attributes. More information about preferring virtual terminal sequences can be found in the **classic functions versus virtual terminal** document.

Font Attributes

The **GetCurrentConsoleFont** function retrieves information about the current console font. The information stored in the **CONSOLE_FONT_INFO** structure includes the width and height of each character in the font.

The **GetConsoleFontSize** function retrieves the size of the font used by the specified console screen buffer.

> **NOTE**
> Using functions to find and manipulate font information is discouraged. It is recommended to operate command-line applications in a font neutral manner to ensure cross-platform compatibility as well as compatibility with host environments that allow the user to customize the font. More information user preferences and host environments including terminals, please see the **ecosystem roadmap**.

Console Modes

5/18/2021 • 2 minutes to read • Edit Online

Associated with each console input buffer is a set of input modes that affects input operations. Similarly, each console screen buffer has a set of output modes that affects output operations. The input modes can be divided into two groups: those that affect the high-level input functions and those that affect the low-level input functions. The output modes only affect applications that use the high-level output functions.

The **GetConsoleMode** function reports the current input mode of a console's input buffer or the current output mode of a screen buffer. The **SetConsoleMode** function sets the current mode of either a console input buffer or a screen buffer. If a console has multiple screen buffers, the output modes of each can be different. An application can change I/O modes at any time. For more information about the console modes that affect high-level and low-level I/O operations, see High-Level Console Modes and Low-Level Console Modes.

A command-line application should expect that other command-line applications may change the console mode at any time and may not restore it to its original form before control is returned. Additionally, we recommend that all command-line applications should capture the initial console mode at startup and attempt to restore it when exiting to ensure minimal impact on other command-line applications attached to the same console.

The **GetConsoleDisplayMode** function reports whether the current console is in full-screen mode.

Console Process Groups

5/18/2021 • 2 minutes to read • Edit Online

When a process uses the **CreateProcess** function to create a new console process, it can specify the **CREATE_NEW_PROCESS_GROUP** flag to make the new process the root process of a console process group. The process group includes all processes that are descendants of the root process.

A process can use the **GenerateConsoleCtrlEvent** function to send a CTRL+C or CTRL+BREAK signal to all processes in a console process group. The signal is only received by those processes in the group that are attached to the same console as the process that called **GenerateConsoleCtrlEvent**.

Window and Screen Buffer Size

5/18/2021 • 2 minutes to read • Edit Online

The size of a screen buffer is expressed in terms of a coordinate grid based on character cells. The width is the number of character cells in each row, and the height is the number of rows. Associated with each screen buffer is a window that determines the size and location of the rectangular portion of the console screen buffer displayed in the console window. A screen buffer's window is defined by specifying the character-cell coordinates of the upper left and lower right cells of the window's rectangle.

> **NOTE**
>
> In the **virtual terminal sequences** world, the size of the window and the size of the screen buffer are fixed to the same value. The terminal handles any scrollback region that would be the equivalent of a console with a screen buffer size larger than its window size. That content belongs to the terminal and is generally no longer a part of the addressable area. For more information, please see our comparison of the **classic console functions versus virtual terminal sequences**.

A screen buffer can be any size, limited only by available memory. The dimensions of a screen buffer's window cannot exceed the corresponding dimensions of either the console screen buffer or the maximum window that can fit on the screen based on the current font size (controlled exclusively by the user).

The **GetConsoleScreenBufferInfo** function returns the following information about a screen buffer and its window:

- The current size of the console screen buffer
- The current location of the window
- The maximum size of the window given the current screen buffer size, the current font size, and the screen size

The **GetLargestConsoleWindowSize** function returns the maximum size of a console's window based on the current font and screen sizes. This size differs from the maximum window size returned by **GetConsoleScreenBufferInfo** in that the console screen buffer size is ignored.

To change a screen buffer's size, use the **SetConsoleScreenBufferSize** function. This function fails if either dimension of the specified size is less than the corresponding dimension of the console's window.

To change the size or location of a screen buffer's window, use the **SetConsoleWindowInfo** function. This function fails if the specified window-corner coordinates exceed the limits of the console screen buffer or the screen. Changing the window size of the active screen buffer changes the size of the console window displayed on the screen.

A process can change its console's input mode to enable window input so that the process is able to receive input when the user changes the console screen buffer size. If an application enables window input, it can use **GetConsoleScreenBufferInfo** to retrieve window and screen buffer size at startup. This information can then be used to determine the way data is displayed in the window. If the user changes the console screen buffer size, the application can respond by changing the way data is displayed. For example, an application can adjust the way text wraps at the end of the line if the number of characters per row changes. If an application does not enable window input, it must either use the inherited window and screen buffer sizes, or set them to the desired size during startup and restore the inherited sizes at exit. For additional information about window input mode, see Low-Level Console Modes.

Console Selection

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

An accessibility application needs information about the user's selection in the console. To retrieve the current console selection, call the GetConsoleSelectionInfo function. The CONSOLE_SELECTION_INFO structure contains information about the selection, such as the anchor, coordinates, and status.

Legacy Console mode

5/18/2021 • 2 minutes to read • Edit Online

Legacy Console mode is a compatibility tool designed to help users of older command-line tools on Windows 10. For any command-line tool that is not displaying or operating correctly in the default Windows 10 console experience, this mode provides a coarse-grained solution to stepping the system back to an older version of the console hosting experience.

Using Legacy Console Mode

To use Legacy Console mode, first open any console hosting window. This is typically done by launching one of the command interpreters CMD or PowerShell.

Right-click on the application title bar and choose the `Properties` menu option. Choose the first tab, `Options`. Then check the box at the bottom of the page describing `Use legacy console`. Press the `OK` button to apply.

The setting can be reverted by returning to the same property sheet menu and unchecking the box then pressing `OK`.

> **NOTE**
> This setting is globally applied to all sessions that start after the preference is changed. Sessions that are already open will not be changed.

Differences between modes

The Console Host team strives to minimize differences between the Legacy and current modes of the console to ensure that as many customers as possible can run the most up-to-date version. If you experience an issue that requires you to use the legacy console that is not documented here, please contact the team on the microsoft/terminal GitHub repository or via the Feedback Hub for assistance.

16-bit applications on 32-bit Windows

Some 16-bit applications on 32-bit Windows use a virtual machine technology to operate called NTVDM. Often these applications use a graphical screen buffering mode in conjunction with the console hosting environment to operate. Only the legacy console experience supports these graphical buffering modes and the additional console API support required to power these applications. The system will automatically select the legacy console environment when one of these applications is launched.

IME Embedding

The legacy Console Host embedded the suggestion portion of the IME inside the hosting window by reserving a line at the bottom of the screen for suggestions. The current Console Host environment instead delegates this activity to the IME subsystem to display an overlay window above the console host with suggestions. In an environment where overlay windows are not possible (like with certain remoting tools), the legacy console host may be required.

API Differences

The major known difference between legacy and current is the implementation of UTF-8. The legacy host has extremely rudimentary and often incorrect support of UTF-8 with code page 65001. The current console host contains incremental improvements release-over-release of Windows 10 to improve this support. Applications that are attempting to rely on predicting "known incorrect" interpretations of UTF-8 from the legacy console will

find themselves receiving different answers as support is improved.

Other differences experienced with APIs should be reported to the microsoft/terminal GitHub repository or via the Feedback Hub for triage and possible remediation.

Pseudoconsoles

5/18/2021 • 2 minutes to read • Edit Online

A *pseudoconsole* is a device type that allows applications to become the host for character-mode applications.

This is in contrast to a typical console session where the operating system will create a hosting window on behalf of the character-mode application to handle graphical output and user input.

With a pseudoconsole, the hosting window is not created. The application that makes the pseudoconsole must become responsible for displaying the graphical output and collecting user input. Alternatively, the information can be relayed further to another application responsible for these activities at a later point in the chain.

This functionality is designed for third-party "terminal window" applications to exist on the platform or for redirection of character-mode activities to a remote "terminal window" session on another machine or even on another platform.

Note that the underlying console session will still be created on behalf of the application requesting the pseudoconsole. All the rules of console sessions still apply including the ability for multiple client character-mode applications to connect to the session.

To provide maximum compatibility with the existing world of pseudoterminal functionality, the information provided over the pseudoconsole channel will always be encoded in UTF-8. This does not affect the codepage or encoding of the client applications that are attached. Translation will happen inside the pseudoconsole system as necessary.

An example for getting started can be found at Creating a Pseudoconsole Session.

Some additional background information on pseudoconsoles can be found at the announcement blog post: Windows Command-Line: Introducing the Windows Pseudo Console (ConPTY).

Console Reference

5/18/2021 • 2 minutes to read • Edit Online

The following sections describe the Console API:

- Console Functions
- Console Structures
- Console WinEvents

Using the Console

5/18/2021 • 2 minutes to read • Edit Online

The following examples demonstrate how to use the console functions:

- Using the high-level input and output functions
- Reading and writing blocks of characters and attributes
- Reading input buffer events
- Clearing the screen
- Scrolling a screen buffer's window
- Scrolling a screen buffer's contents
- Registering a control handler function

High-Level Console Input and Output Functions

5/18/2021 • 2 minutes to read • Edit Online

The **ReadFile** and **WriteFile** functions, or the **ReadConsole** and **WriteConsole** functions, enable an application to read console input and write console output as a stream of characters. **ReadConsole** and **WriteConsole** behave exactly like **ReadFile** and **WriteFile** except that they can be used either as wide-character functions (in which text arguments must use Unicode) or as ANSI functions (in which text arguments must use characters from the Windows character set). Applications that need to maintain a single set of sources to support either Unicode or the ANSI character set should use **ReadConsole** and **WriteConsole**.

ReadConsole and **WriteConsole** can only be used with console handles; **ReadFile** and **WriteFile** can be used with other handles (such as files or pipes). **ReadConsole** and **WriteConsole** fail if used with a standard handle that has been redirected and is no longer a console handle.

To get keyboard input, a process can use **ReadFile** or **ReadConsole** with a handle to the console's input buffer, or it can use **ReadFile** to read input from a file or a pipe if `STDIN` has been redirected. These functions only return keyboard events that can be translated into ANSI or Unicode characters. The input that can be returned includes control key combinations. The functions do not return keyboard events involving the function keys or arrow keys. Input events generated by mouse, window, focus, or menu input are discarded.

If line input mode is enabled (the default mode), **ReadFile** and **ReadConsole** do not return to the calling application until the ENTER key is pressed. If line input mode is disabled, the functions do not return until at least one character is available. In either mode, all available characters are read until either no more keys are available or the specified number of characters has been read. Unread characters are buffered until the next read operation. The functions report the total number of characters actually read. If echo input mode is enabled, characters read by these functions are written to the active screen buffer at the current cursor position.

A process can use **WriteFile** or **WriteConsole** to write to either an active or inactive screen buffer, or it can use **WriteFile** to write to a file or a pipe if STDOUT has been redirected. Processed output mode and wrap at EOL output mode control the way characters are written or echoed to a screen buffer.

Characters written by **WriteFile** or **WriteConsole**, or echoed by **ReadFile** or **ReadConsole**, are inserted in a screen buffer at the current cursor position. As each character is written, the cursor position advances to the next character cell; however, the behavior at the end of a row depends on the console screen buffer's wrap at EOL output mode.

Further detail about the position of the cursor can be found through **virtual terminals sequences**, specifically in the **query state** category for finding the current position and the **cursor positioning** category for setting the current position. Alternatively, an application can use the **GetConsoleScreenBufferInfo** function to determine the current cursor position and the **SetConsoleCursorPosition** function to set the cursor position. However, the **virtual terminal sequences** mechanism is preferred for all new and ongoing development. More details on the strategy behind this decision can be found in the **classic functions versus virtual terminal** and **ecosystem roadmap** documentation.

For an example that uses the high-level console I/O functions, see Using the High-Level Input and Output Functions.

Using the High-Level Input and Output Functions

5/18/2021 • 4 minutes to read • Edit Online

> **IMPORTANT**
>
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

The following example uses the high-level console I/O functions for console I/O. For more information about the high-level console I/O functions, see High-Level Console I/O.

The example assumes that the default I/O modes are in effect initially for the first calls to the **ReadFile** and **WriteFile** functions. Then the input mode is changed to turn offline input mode and echo input mode for the second calls to **ReadFile** and **WriteFile**. The **SetConsoleTextAttribute** function is used to set the colors in which subsequently written text will be displayed. Before exiting, the program restores the original console input mode and color attributes.

The example's `NewLine` function is used when line input mode is disabled. It handles carriage returns by moving the cursor position to the first cell of the next row. If the cursor is already in the last row of the console screen buffer, the contents of the console screen buffer are scrolled up one line.

```
#include <windows.h>

void NewLine(void);
void ScrollScreenBuffer(HANDLE, INT);

HANDLE hStdout, hStdin;
CONSOLE_SCREEN_BUFFER_INFO csbiInfo;

int main(void)
{
    LPSTR lpszPrompt1 = "Type a line and press Enter, or q to quit: ";
    LPSTR lpszPrompt2 = "Type any key, or q to quit: ";
    CHAR chBuffer[256];
    DWORD cRead, cWritten, fdwMode, fdwOldMode;
    WORD wOldColorAttrs;

    // Get handles to STDIN and STDOUT.

    hStdin = GetStdHandle(STD_INPUT_HANDLE);
    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hStdin == INVALID_HANDLE_VALUE ||
        hStdout == INVALID_HANDLE_VALUE)
    {
        MessageBox(NULL, TEXT("GetStdHandle"), TEXT("Console Error"),
            MB_OK);
        return 1;
    }

    // Save the current text colors.

    if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        MessageBox(NULL, TEXT("GetConsoleScreenBufferInfo"),
```

```
        TEXT("Console Error"), MB_OK);
    return 1;
}

wOldColorAttrs = csbiInfo.wAttributes;

// Set the text attributes to draw red text on black background.

if (! SetConsoleTextAttribute(hStdout, FOREGROUND_RED |
        FOREGROUND_INTENSITY))
{
    MessageBox(NULL, TEXT("SetConsoleTextAttribute"),
        TEXT("Console Error"), MB_OK);
    return 1;
}

// Write to STDOUT and read from STDIN by using the default
// modes. Input is echoed automatically, and ReadFile
// does not return until a carriage return is typed.
//
// The default input modes are line, processed, and echo.
// The default output modes are processed and wrap at EOL.

while (1)
{
    if (! WriteFile(
        hStdout,               // output handle
        lpszPrompt1,           // prompt string
        lstrlenA(lpszPrompt1), // string length
        &cWritten,             // bytes written
        NULL) )                // not overlapped
    {
        MessageBox(NULL, TEXT("WriteFile"), TEXT("Console Error"),
            MB_OK);
        return 1;
    }

    if (! ReadFile(
        hStdin,    // input handle
        chBuffer,  // buffer to read into
        255,       // size of buffer
        &cRead,    // actual bytes read
        NULL) )    // not overlapped
    break;
    if (chBuffer[0] == 'q') break;
}

// Turn off the line input and echo input modes

if (! GetConsoleMode(hStdin, &fdwOldMode))
{
   MessageBox(NULL, TEXT("GetConsoleMode"), TEXT("Console Error"),
       MB_OK);
   return 1;
}

fdwMode = fdwOldMode &
    ~(ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT);
if (! SetConsoleMode(hStdin, fdwMode))
{
   MessageBox(NULL, TEXT("SetConsoleMode"), TEXT("Console Error"),
       MB_OK);
   return 1;
}

// ReadFile returns when any input is available.
// WriteFile is used to echo input.

NewLine();
```

```
    while (1)
    {
        if (! WriteFile(
            hStdout,               // output handle
            lpszPrompt2,           // prompt string
            lstrlenA(lpszPrompt2), // string length
            &cWritten,             // bytes written
            NULL) )                // not overlapped
        {
            MessageBox(NULL, TEXT("WriteFile"), TEXT("Console Error"),
                MB_OK);
            return 1;
        }

        if (! ReadFile(hStdin, chBuffer, 1, &cRead, NULL))
            break;
        if (chBuffer[0] == '\r')
            NewLine();
        else if (! WriteFile(hStdout, chBuffer, cRead,
            &cWritten, NULL)) break;
        else
            NewLine();
        if (chBuffer[0] == 'q') break;
    }

    // Restore the original console mode.

    SetConsoleMode(hStdin, fdwOldMode);

    // Restore the original text colors.

    SetConsoleTextAttribute(hStdout, wOldColorAttrs);

    return 0;
}

// The NewLine function handles carriage returns when the processed
// input mode is disabled. It gets the current cursor position
// and resets it to the first cell of the next row.

void NewLine(void)
{
    if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        MessageBox(NULL, TEXT("GetConsoleScreenBufferInfo"),
            TEXT("Console Error"), MB_OK);
        return;
    }

    csbiInfo.dwCursorPosition.X = 0;

    // If it is the last line in the screen buffer, scroll
    // the buffer up.

    if ((csbiInfo.dwSize.Y-1) == csbiInfo.dwCursorPosition.Y)
    {
        ScrollScreenBuffer(hStdout, 1);
    }

    // Otherwise, advance the cursor to the next line.

    else csbiInfo.dwCursorPosition.Y += 1;

    if (! SetConsoleCursorPosition(hStdout,
        csbiInfo.dwCursorPosition))
    {
        MessageBox(NULL, TEXT("SetConsoleCursorPosition"),
            TEXT("Console Error"), MB_OK);
```

```
            TEXT("Console Error"), MB_OK);
        return;
    }
}

void ScrollScreenBuffer(HANDLE h, INT x)
{
    SMALL_RECT srctScrollRect, srctClipRect;
    CHAR_INFO chiFill;
    COORD coordDest;

    srctScrollRect.Left = 0;
    srctScrollRect.Top = 1;
    srctScrollRect.Right = csbiInfo.dwSize.X - (SHORT)x;
    srctScrollRect.Bottom = csbiInfo.dwSize.Y - (SHORT)x;

    // The destination for the scroll rectangle is one row up.

    coordDest.X = 0;
    coordDest.Y = 0;

    // The clipping rectangle is the same as the scrolling rectangle.
    // The destination row is left unchanged.

    srctClipRect = srctScrollRect;

    // Set the fill character and attributes.

    chiFill.Attributes = FOREGROUND_RED|FOREGROUND_INTENSITY;
    chiFill.Char.AsciiChar = (char)' ';

    // Scroll up one line.

    ScrollConsoleScreenBuffer(
        h,               // screen buffer handle
        &srctScrollRect, // scrolling rectangle
        &srctClipRect,   // clipping rectangle
        coordDest,       // top left destination cell
        &chiFill);       // fill character and color
}
```

High-Level Console Modes

5/18/2021 • 6 minutes to read • Edit Online

The behavior of the high-level console functions is affected by the console input and output modes. All of the following console input modes are enabled for a console's input buffer when a console is created:

- Line input mode
- Processed input mode
- Echo input mode

Both of the following console output modes are enabled for a console screen buffer when it is created:

- Processed output mode
- Wrapping at EOL output mode

All three input modes, along with processed output mode, are designed to work together. It is best to either enable or disable all of these modes as a group. When all are enabled, the application is said to be in "cooked" mode, which means that most of the processing is handled for the application. When all are disabled, the application is in "raw" mode, which means that input is unfiltered and any processing is left to the application.

An application can use the GetConsoleMode function to determine the current mode of a console's input buffer or screen buffer. You can enable or disable any of these modes by using the following values in the SetConsoleMode function. Note that setting the output mode of one screen buffer does not affect the output mode of other screen buffers.

If the *hConsoleHandle* parameter is an input handle, the mode can be one or more of the following values. When a console is created, all input modes except **ENABLE_WINDOW_INPUT** and **ENABLE_VIRTUAL_TERMINAL_INPUT** are enabled by default.

VALUE	MEANING
ENABLE_ECHO_INPUT 0x0004	Characters read by the ReadFile or ReadConsole function are written to the active screen buffer as they are typed into the console. This mode can be used only if the **ENABLE_LINE_INPUT** mode is also enabled.
ENABLE_INSERT_MODE 0x0020	When enabled, text entered in a console window will be inserted at the current cursor location and all text following that location will not be overwritten. When disabled, all following text will be overwritten.
ENABLE_LINE_INPUT 0x0002	The ReadFile or ReadConsole function returns only when a carriage return character is read. If this mode is disabled, the functions return when one or more characters are available.
ENABLE_MOUSE_INPUT 0x0010	If the mouse pointer is within the borders of the console window and the window has the keyboard focus, mouse events generated by mouse movement and button presses are placed in the input buffer. These events are discarded by ReadFile or ReadConsole, even when this mode is enabled.

VALUE	MEANING
ENABLE_PROCESSED_INPUT 0x0001	CTRL+C is processed by the system and is not placed in the input buffer. If the input buffer is being read by **ReadFile** or **ReadConsole**, other control keys are processed by the system and are not returned in the **ReadFile** or **ReadConsole** buffer. If the **ENABLE_LINE_INPUT** mode is also enabled, backspace, carriage return, and line feed characters are handled by the system.
ENABLE_QUICK_EDIT_MODE 0x0040	This flag enables the user to use the mouse to select and edit text. To enable this mode, use `ENABLE_QUICK_EDIT_MODE | ENABLE_EXTENDED_FLAGS`. To disable this mode, use **ENABLE_EXTENDED_FLAGS** without this flag.
ENABLE_WINDOW_INPUT 0x0008	User interactions that change the size of the console screen buffer are reported in the console's input buffer. Information about these events can be read from the input buffer by applications using the **ReadConsoleInput** function, but not by those using **ReadFile** or **ReadConsole**.
ENABLE_VIRTUAL_TERMINAL_INPUT 0x0200	Setting this flag directs the Virtual Terminal processing engine to convert user input received by the console window into **Console Virtual Terminal Sequences** that can be retrieved by a supporting application through **WriteFile** or **WriteConsole** functions. The typical usage of this flag is intended in conjunction with ENABLE_VIRTUAL_TERMINAL_PROCESSING on the output handle to connect to an application that communicates exclusively via virtual terminal sequences.

If the *hConsoleHandle* parameter is a screen buffer handle, the mode can be one or more of the following values. When a screen buffer is created, both output modes are enabled by default.

VALUE	MEANING
ENABLE_PROCESSED_OUTPUT 0x0001	Characters written by the **WriteFile** or **WriteConsole** function or echoed by the **ReadFile** or **ReadConsole** function are parsed for ASCII control sequences, and the correct action is performed. Backspace, tab, bell, carriage return, and line feed characters are processed.
ENABLE_WRAP_AT_EOL_OUTPUT 0x0002	When writing with **WriteFile** or **WriteConsole** or echoing with **ReadFile** or **ReadConsole**, the cursor moves to the beginning of the next row when it reaches the end of the current row. This causes the rows displayed in the console window to scroll up automatically when the cursor advances beyond the last row in the window. It also causes the contents of the console screen buffer to scroll up (../discarding the top row of the console screen buffer) when the cursor advances beyond the last row in the console screen buffer. If this mode is disabled, the last character in the row is overwritten with any subsequent characters.

VALUE	MEANING
ENABLE_VIRTUAL_TERMINAL_PROCESSING 0x0004	When writing with **WriteFile** or **WriteConsole**, characters are parsed for VT100 and similar control character sequences that control cursor movement, color/font mode, and other operations that can also be performed via the existing Console APIs. For more information, see **Console Virtual Terminal Sequences**.
DISABLE_NEWLINE_AUTO_RETURN 0x0008	When writing with **WriteFile** or **WriteConsole**, this adds an additional state to end-of-line wrapping that can delay the cursor move and buffer scroll operations. Normally when **ENABLE_WRAP_AT_EOL_OUTPUT** is set and text reaches the end of the line, the cursor will immediately move to the next line and the contents of the buffer will scroll up by one line. In contrast with this flag set, the scroll operation and cursor move is delayed until the next character arrives. The written character will be printed in the final position on the line and the cursor will remain above this character as if **ENABLE_WRAP_AT_EOL_OUTPUT** was off, but the next printable character will be printed as if **ENABLE_WRAP_AT_EOL_OUTPUT** is on. No overwrite will occur. Specifically, the cursor quickly advances down to the following line, a scroll is performed if necessary, the character is printed, and the cursor advances one more position. The typical usage of this flag is intended in conjunction with setting **ENABLE_VIRTUAL_TERMINAL_PROCESSING** to better emulate a terminal emulator where writing the final character on the screen (../in the bottom right corner) without triggering an immediate scroll is the desired behavior.
ENABLE_LVB_GRID_WORLDWIDE 0x0010	The APIs for writing character attributes including **WriteConsoleOutput** and **WriteConsoleOutputAttribute** allow the usage of flags from **character attributes** to adjust the color of the foreground and background of text. Additionally, a range of DBCS flags was specified with the COMMON_LVB prefix. Historically, these flags only functioned in DBCS code pages for Chinese, Japanese, and Korean languages. With exception of the leading byte and trailing byte flags, the remaining flags describing line drawing and reverse video (../swap foreground and background colors) can be useful for other languages to emphasize portions of output. Setting this console mode flag will allow these attributes to be used in every code page on every language. It is off by default to maintain compatibility with known applications that have historically taken advantage of the console ignoring these flags on non-CJK machines to store bits in these fields for their own purposes or by accident. Note that using the ENABLE_VIRTUAL_TERMINAL_PROCESSING mode can result in LVB grid and reverse video flags being set while this flag is still off if the attached application requests underlining or inverse video via **Console Virtual Terminal Sequences**.

High-Level Console I/O

5/18/2021 • 2 minutes to read • Edit Online

The high-level I/O functions provide a simple way to read a stream of characters from console input or to write a stream of characters to console output. A high-level read operation gets input characters from a console's input buffer and stores them in a specified buffer. A high-level write operation takes characters from a specified buffer and writes them to a screen buffer at the current cursor location, advancing the cursor as each character is written.

High-level I/O gives you a choice between the **ReadFile** and **WriteFile** functions and the **ReadConsole** and **WriteConsole** functions. They are identical, except for two important differences. The console functions support the use of either Unicode characters or the ANSI character set through the A and W variants of each function; the file I/O functions do not support Unicode except for UTF-8 set with the `CP_UTF8` constant on the **SetConsoleCP** and **SetConsoleOutputCP** functions prior to use. Also, the file I/O functions can be used to access files, pipes, and serial communications devices; the console functions can only be used with console handles. This distinction is important if an application relies on standard handles that may have been redirected.

When using either set of high-level functions, an application can control the text and background colors used to display characters subsequently written to a screen buffer with the preferred mechanism being via **virtual terminal sequences**. An application can also use the console modes that affect high-level console I/O to enable or disable the following properties:

- Echoing of keyboard input to the active screen buffer
- Line input, in which a read operation does not return until the ENTER key is pressed
- Automatic processing of keyboard input to handle carriage returns, CTRL+C, and other input details
- Automatic processing of output to handle line wrapping, carriage returns, backspaces, and other output details

For more information, see the following topics:

- Console Modes
- High-Level Console Modes
- High-Level Console Input and Output Functions
- Classic APIs Versus Virtual Terminal Sequences

Low-Level Console Input Functions

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

A low-level console input functions buffer contains input records that can include information about keyboard, mouse, buffer-resizing, focus, and menu events. The low-level functions provide direct access to the input buffer, unlike the high-level functions that filter and process the input buffer's data, discarding all but keyboard input.

There are five low-level functions for accessing a console's input buffer:

- ReadConsoleInput
- PeekConsoleInput
- GetNumberOfConsoleInputEvents
- WriteConsoleInput
- FlushConsoleInputBuffer

The ReadConsoleInput, PeekConsoleInput, and WriteConsoleInput functions use the INPUT_RECORD structure to read from or write to an input buffer.

Following are descriptions of the low-level console input functions.

FUNCTION	DESCRIPTION
ReadConsoleInput	Reads and removes input records from an input buffer. The function does not return until at least one record is available to be read. Then all available records are transferred to the buffer of the calling process until either no more records are available or the specified number of records has been read. Unread records remain in the input buffer for the next read operation. The function reports the total number of records that have been read. For an example that uses **ReadConsoleInput**, see Reading Input Buffer Events.
PeekConsoleInput	Reads without removing the pending input records in an input buffer. All available records up to the specified number are copied into the buffer of the calling process. If no records are available, the function returns immediately. The function reports the total number of records that have been read.
GetNumberOfConsoleInputEvents	Determines the number of unread input records in an input buffer.

FUNCTION	DESCRIPTION
WriteConsoleInput	Places input records into the input buffer behind any pending records in the buffer. The input buffer grows dynamically, if necessary, to hold as many records as are written. To use this function, the specified input buffer handle must have the GENERIC_WRITE access right.
FlushConsoleInputBuffer	Discards all unread events in the input buffer. To use this function, the specified input buffer handle must have the GENERIC_WRITE access right.

A thread of an application's process can perform a wait operation to wait for input to be available in an input buffer. To initiate a wait operation, specify a handle to the input buffer in a call to any of the wait functions. These functions can return when the state of one or more objects is signaled. The state of a console input handle becomes signaled when there are unread records in its input buffer. The state is reset to non-signaled when the input buffer becomes empty. If there is no input available, the calling thread enters an efficient wait state, consuming very little processor time while waiting for the conditions of the wait operation to be satisfied.

Low-Level Console Output Functions

5/18/2021 • 3 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

The low-level console output functions provide direct access to the character cells of a screen buffer. One set of functions reads from or writes to consecutive cells beginning at any location in the console screen buffer. Another set of functions reads from or writes to rectangular blocks of cells.

The following functions read from or write to a specified number of consecutive character cells in a screen buffer, beginning with a specified cell.

FUNCTION	DESCRIPTION
ReadConsoleOutputCharacter	Copies a string of Unicode or ANSI characters from a screen buffer.
WriteConsoleOutputCharacter	Writes a string of Unicode or ANSI characters to a screen buffer.
ReadConsoleOutputAttribute	Copies a string of text and background color attributes from a screen buffer.
WriteConsoleOutputAttribute	Writes a string of text and background color attributes to a screen buffer.
FillConsoleOutputCharacter	Writes a single Unicode or ANSI character to a specified number of consecutive cells in a screen buffer.
FillConsoleOutputAttribute	Writes a text and background color attribute combination to a specified number of consecutive cells in a screen buffer.

For all of these functions, when the last cell of a row is encountered, reading or writing wraps around to the first cell of the next row. When the end of the last row of the console screen buffer is encountered, the write functions discard all unwritten characters or attributes, and the read functions report the number of characters or attributes actually written.

The following functions read from or write to rectangular blocks of character cells at a specified location in a screen buffer.

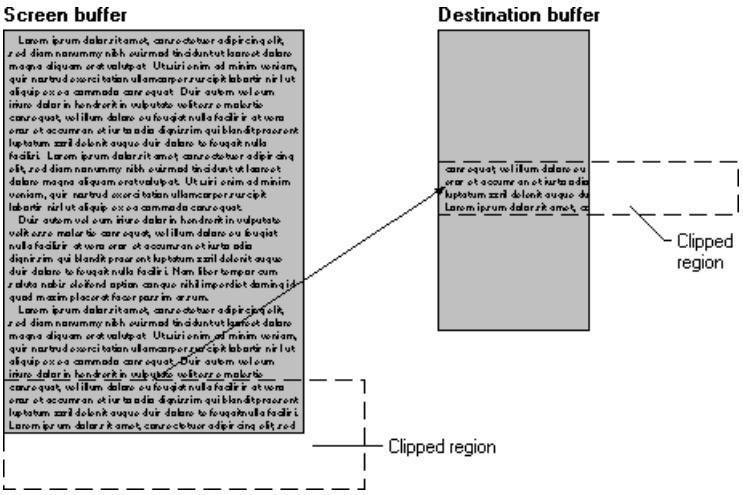
FUNCTION	DESCRIPTION
ReadConsoleOutput	Copies character and color data from a specified block of screen buffer cells into a given block in a destination buffer.

FUNCTION	DESCRIPTION
WriteConsoleOutput	Writes character and color data to a specified block of screen buffer cells from a given block in a source buffer.

These functions treat screen buffers and source or destination buffers as two-dimensional arrays of CHAR_INFO structures (containing character and color attribute data for each cell). The functions specify the width and height, in character cells, of the source or destination buffer, and the pointer to the buffer is treated as a pointer to the origin cell (0,0) of the two-dimensional array. The functions use a SMALL_RECT structure to specify which rectangle to access in the console screen buffer, and the coordinates of the upper left cell in the source or destination buffer determine the location of the corresponding rectangle in that buffer.

These functions automatically clip the specified screen buffer rectangle to fit within the boundaries of the console screen buffer. For example, if the rectangle specifies lower right coordinates that are (column 100, row 50) and the console screen buffer is only 80 columns wide, the coordinates are clipped so that they are (column 79, row 50). Similarly, this adjusted rectangle is again clipped to fit within the boundaries of the source or destination buffer. The screen buffer coordinates of the actual rectangle that was read from or written to are specified. For an example that uses these functions, see Reading and Writing Blocks of Characters and Attributes.

The illustration shows a ReadConsoleOutput operation where clipping occurs when the block is read from the console screen buffer, and again when the block is copied into the destination buffer. The function reports the actual screen buffer rectangle that it copied from.



Low-Level Console I/O

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

The low-level console I/O functions expand an application's control over console I/O by enabling direct access to a console's input and screen buffers. These functions enable an application to perform the following tasks:

- Receive input about mouse and buffer-resizing events
- Receive extended information about keyboard input events
- Write input records to the input buffer
- Read input records without removing them from the input buffer
- Determine the number of pending events in the input buffer
- Flush the input buffer
- Read and write strings of Unicode or ANSI characters at a specified location in a screen buffer
- Read and write strings of text and background color attributes at a specified screen buffer location
- Read and write rectangular blocks of character and color data at a specified screen buffer location
- Write a single Unicode or ANSI character, or a text and background color attribute combination, to a specified number of consecutive cells beginning at a specified screen buffer location

For more information, see the following topics:

- Console Modes
- Low-Level Console Modes
- Low-Level Console Input Functions
- Low-Level Console Output Functions

Low-Level Console Modes

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

The types of input events reported in a console's input buffer depend on the console's mouse and window input modes. The console's processed input mode determines how the system handles the CTRL+C key combination. To set or retrieve the state of a console's input modes, an application can specify a console input buffer handle in a call to the SetConsoleMode or GetConsoleMode function. The following modes are used with console input handles.

MODE	DESCRIPTION
ENABLE_MOUSE_INPUT	Controls whether mouse events are reported in the input buffer. By default, mouse input is enabled and window input is disabled. Changing either of these modes affects only input that occurs after the mode is set; pending mouse or window events in the input buffer are not flushed. The mouse pointer is displayed regardless of the mouse mode.
ENABLE_WINDOW_INPUT	Controls whether buffer-resizing events are reported in the input buffer. By default, mouse input is enabled and window input is disabled. Changing either of these modes affects only input that occurs after the mode is set; pending mouse or window events in the input buffer are not flushed. The mouse pointer is displayed regardless of the mouse mode.
ENABLE_PROCESSED_INPUT	Controls the processing of input for applications using the high-level console I/O functions. However, if processed input mode is enabled, the CTRL+C key combination is not reported in the console's input buffer. Instead, it is passed on to the appropriate control handler function. For more information about control handlers, see Console Control Handlers.

The output modes of a screen buffer do not affect the behavior of the low-level output functions.

Reading and Writing Blocks of Characters and Attributes

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

The **ReadConsoleOutput** function copies a rectangular block of character and color attribute data from a console screen buffer into a destination buffer. The function treats the destination buffer as a two-dimensional array of **CHAR_INFO** structures. Similarly, the **WriteConsoleOutput** function copies a rectangular block of character and color attribute data from a source buffer to a console screen buffer. For more information about reading from or writing to rectangular blocks of screen buffer cells, see Input and Output Methods.

The following example uses the **CreateConsoleScreenBuffer** function to create a new screen buffer. After the **SetConsoleActiveScreenBuffer** function makes this the active screen buffer, a block of characters and color attributes is copied from the top two rows of the STDOUT screen buffer into a temporary buffer. The data is then copied from the temporary buffer into the new active screen buffer. When the application is finished using the new screen buffer, it calls **SetConsoleActiveScreenBuffer** to restore the original STDOUT screen buffer.

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

int main(void)
{
    HANDLE hStdout, hNewScreenBuffer;
    SMALL_RECT srctReadRect;
    SMALL_RECT srctWriteRect;
    CHAR_INFO chiBuffer[160]; // [2][80];
    COORD coordBufSize;
    COORD coordBufCoord;
    BOOL fSuccess;

    // Get a handle to the STDOUT screen buffer to copy from and
    // create a new screen buffer to copy to.

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    hNewScreenBuffer = CreateConsoleScreenBuffer(
       GENERIC_READ |           // read/write access
       GENERIC_WRITE,
       FILE_SHARE_READ |
       FILE_SHARE_WRITE,        // shared
       NULL,                    // default security attributes
       CONSOLE_TEXTMODE_BUFFER, // must be TEXTMODE
       NULL);                   // reserved; must be NULL
    if (hStdout == INVALID_HANDLE_VALUE ||
            hNewScreenBuffer == INVALID_HANDLE_VALUE)
    {
        printf("CreateConsoleScreenBuffer failed - (%d)\n", GetLastError());
        return 1;
    }
```

```
    // Make the new screen buffer the active screen buffer.

    if (! SetConsoleActiveScreenBuffer(hNewScreenBuffer) )
    {
        printf("SetConsoleActiveScreenBuffer failed - (%d)\n", GetLastError());
        return 1;
    }

    // Set the source rectangle.

    srctReadRect.Top = 0;    // top left: row 0, col 0
    srctReadRect.Left = 0;
    srctReadRect.Bottom = 1; // bot. right: row 1, col 79
    srctReadRect.Right = 79;

    // The temporary buffer size is 2 rows x 80 columns.

    coordBufSize.Y = 2;
    coordBufSize.X = 80;

    // The top left destination cell of the temporary buffer is
    // row 0, col 0.

    coordBufCoord.X = 0;
    coordBufCoord.Y = 0;

    // Copy the block from the screen buffer to the temp. buffer.

    fSuccess = ReadConsoleOutput(
       hStdout,        // screen buffer to read from
       chiBuffer,      // buffer to copy into
       coordBufSize,   // col-row size of chiBuffer
       coordBufCoord,  // top left dest. cell in chiBuffer
       &srctReadRect); // screen buffer source rectangle
    if (! fSuccess)
    {
        printf("ReadConsoleOutput failed - (%d)\n", GetLastError());
        return 1;
    }

    // Set the destination rectangle.

    srctWriteRect.Top = 10;    // top lt: row 10, col 0
    srctWriteRect.Left = 0;
    srctWriteRect.Bottom = 11; // bot. rt: row 11, col 79
    srctWriteRect.Right = 79;

    // Copy from the temporary buffer to the new screen buffer.

    fSuccess = WriteConsoleOutput(
        hNewScreenBuffer, // screen buffer to write to
        chiBuffer,        // buffer to copy from
        coordBufSize,     // col-row size of chiBuffer
        coordBufCoord,    // top left src cell in chiBuffer
        &srctWriteRect);  // dest. screen buffer rectangle
    if (! fSuccess)
    {
        printf("WriteConsoleOutput failed - (%d)\n", GetLastError());
        return 1;
    }
    Sleep(5000);

    // Restore the original active screen buffer.

    if (! SetConsoleActiveScreenBuffer(hStdout))
    {
        printf("SetConsoleActiveScreenBuffer failed - (%d)\n", GetLastError());
        return 1;
```

```
    }

    return 0;
}
```

Reading Input Buffer Events

5/18/2021 • 2 minutes to read • Edit Online

The **ReadConsoleInput** function can be used to directly access a console's input buffer. When a console is created, mouse input is enabled and window input is disabled. To ensure that the process receives all types of events, this example uses the **SetConsoleMode** function to enable window and mouse input. Then it goes into a loop that reads and handles 100 console input events. For example, the message "Keyboard event" is displayed when the user presses a key and the message "Mouse event" is displayed when the user interacts with the mouse.

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

HANDLE hStdin;
DWORD fdwSaveOldMode;

VOID ErrorExit(LPSTR);
VOID KeyEventProc(KEY_EVENT_RECORD);
VOID MouseEventProc(MOUSE_EVENT_RECORD);
VOID ResizeEventProc(WINDOW_BUFFER_SIZE_RECORD);

int main(VOID)
{
    DWORD cNumRead, fdwMode, i;
    INPUT_RECORD irInBuf[128];
    int counter=0;

    // Get the standard input handle.

    hStdin = GetStdHandle(STD_INPUT_HANDLE);
    if (hStdin == INVALID_HANDLE_VALUE)
        ErrorExit("GetStdHandle");

    // Save the current input mode, to be restored on exit.

    if (! GetConsoleMode(hStdin, &fdwSaveOldMode) )
        ErrorExit("GetConsoleMode");

    // Enable the window and mouse input events.

    fdwMode = ENABLE_WINDOW_INPUT | ENABLE_MOUSE_INPUT;
    if (! SetConsoleMode(hStdin, fdwMode) )
        ErrorExit("SetConsoleMode");

    // Loop to read and handle the next 100 input events.

    while (counter++ <= 100)
    {
        // Wait for the events.

        if (! ReadConsoleInput(
                hStdin,      // input buffer handle
                irInBuf,     // buffer to read into
                128,         // size of read buffer
                &cNumRead) ) // number of records read
            ErrorExit("ReadConsoleInput");

        // Dispatch the events to the appropriate handler.

        for (i = 0; i < cNumRead; i++)
        {
```

```
        {
            switch(irInBuf[i].EventType)
            {
                case KEY_EVENT: // keyboard input
                    KeyEventProc(irInBuf[i].Event.KeyEvent);
                    break;

                case MOUSE_EVENT: // mouse input
                    MouseEventProc(irInBuf[i].Event.MouseEvent);
                    break;

                case WINDOW_BUFFER_SIZE_EVENT: // scrn buf. resizing
                    ResizeEventProc( irInBuf[i].Event.WindowBufferSizeEvent );
                    break;

                case FOCUS_EVENT:  // disregard focus events

                case MENU_EVENT:   // disregard menu events
                    break;

                default:
                    ErrorExit("Unknown event type");
                    break;
            }
        }
    }

    // Restore input mode on exit.

    SetConsoleMode(hStdin, fdwSaveOldMode);

    return 0;
}

VOID ErrorExit (LPSTR lpszMessage)
{
    fprintf(stderr, "%s\n", lpszMessage);

    // Restore input mode on exit.

    SetConsoleMode(hStdin, fdwSaveOldMode);

    ExitProcess(0);
}

VOID KeyEventProc(KEY_EVENT_RECORD ker)
{
    printf("Key event: ");

    if(ker.bKeyDown)
        printf("key pressed\n");
    else printf("key released\n");
}

VOID MouseEventProc(MOUSE_EVENT_RECORD mer)
{
#ifndef MOUSE_HWHEELED
#define MOUSE_HWHEELED 0x0008
#endif
    printf("Mouse event: ");

    switch(mer.dwEventFlags)
    {
        case 0:

            if(mer.dwButtonState == FROM_LEFT_1ST_BUTTON_PRESSED)
            {
                printf("left button press \n");
            }
            else if(mer.dwButtonState == RIGHTMOST_BUTTON_PRESSED)
```

```
            else if(mer.dwButtonState == RIGHTMOST_BUTTON_PRESSED)
            {
                printf("right button press \n");
            }
            else
            {
                printf("button press\n");
            }
            break;
        case DOUBLE_CLICK:
            printf("double click\n");
            break;
        case MOUSE_HWHEELED:
            printf("horizontal mouse wheel\n");
            break;
        case MOUSE_MOVED:
            printf("mouse moved\n");
            break;
        case MOUSE_WHEELED:
            printf("vertical mouse wheel\n");
            break;
        default:
            printf("unknown\n");
            break;
    }
}

VOID ResizeEventProc(WINDOW_BUFFER_SIZE_RECORD wbsr)
{
    printf("Resize event\n");
    printf("Console screen buffer is %d columns by %d rows.\n", wbsr.dwSize.X, wbsr.dwSize.Y);
}
```

Clearing the Screen

9/16/2021 • 3 minutes to read • Edit Online

There are three ways to clear the screen in a console application.

Example 1

> **TIP**
> This is the recommended method using **virtual terminal sequences** for all new development. For more information, see the discussion of **classic console APIs versus virtual terminal sequences**.

The first method is to set your application up for virtual terminal output sequences and then call the "clear screen" command.

```
#include <windows.h>

int main(void)
{
    HANDLE hStdOut;

    hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);

    // Fetch existing console mode so we correctly add a flag and not turn off others
    DWORD mode = 0;
    if (!GetConsoleMode(hStdOut, &mode))
    {
        return ::GetLastError();
    }

    // Hold original mode to restore on exit to be cooperative with other command-line apps.
    const DWORD originalMode = mode;
    mode |= ENABLE_VIRTUAL_TERMINAL_PROCESSING;

    // Try to set the mode.
    if (!SetConsoleMode(hStdOut, mode))
    {
        return ::GetLastError();
    }

    // Write the sequence for clearing the display.
    DWORD written = 0;
    PCWSTR sequence = L"\x1b[2J";
    if (!WriteConsoleW(hStdOut, sequence, (DWORD)wcslen(sequence), &written, NULL))
    {
        // If we fail, try to restore the mode on the way out.
        SetConsoleMode(hStdOut, originalMode);
        return ::GetLastError();
    }

    // To also clear the scroll back, emit L"\x1b[3J" as well.
    // 2J only clears the visible window and 3J only clears the scroll back.

    // Restore the mode on the way out to be nice to other command-line applications.
    SetConsoleMode(hStdOut, originalMode);

    return 0;
}
```

You can find additional variations on this command in the virtual terminal sequences documentation on Erase In Display.

Example 2

The second method is to write a function to scroll the contents of the screen or buffer and set a fill for the revealed space.

This matches the behavior of the command prompt `cmd.exe`.

```
#include <windows.h>

void cls(HANDLE hConsole)
{
    CONSOLE_SCREEN_BUFFER_INFO csbi;
    SMALL_RECT scrollRect;
    COORD scrollTarget;
    CHAR_INFO fill;

    // Get the number of character cells in the current buffer.
    if (!GetConsoleScreenBufferInfo(hConsole, &csbi))
    {
        return;
    }

    // Scroll the rectangle of the entire buffer.
    scrollRect.Left = 0;
    scrollRect.Top = 0;
    scrollRect.Right = csbi.dwSize.X;
    scrollRect.Bottom = csbi.dwSize.Y;

    // Scroll it upwards off the top of the buffer with a magnitude of the entire height.
    scrollTarget.X = 0;
    scrollTarget.Y = (SHORT)(0 - csbi.dwSize.Y);

    // Fill with empty spaces with the buffer's default text attribute.
    fill.Char.UnicodeChar = TEXT(' ');
    fill.Attributes = csbi.wAttributes;

    // Do the scroll
    ScrollConsoleScreenBuffer(hConsole, &scrollRect, NULL, scrollTarget, &fill);

    // Move the cursor to the top left corner too.
    csbi.dwCursorPosition.X = 0;
    csbi.dwCursorPosition.Y = 0;

    SetConsoleCursorPosition(hConsole, csbi.dwCursorPosition);
}

int main(void)
{
    HANDLE hStdout;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

    cls(hStdout);

    return 0;
}
```

Example 3

The third method is to write a function to programmatically clear the screen using the FillConsoleOutputCharacter and FillConsoleOutputAttribute functions.

The following sample code demonstrates this technique.

```
#include <windows.h>

void cls(HANDLE hConsole)
{
    COORD coordScreen = { 0, 0 };    // home for the cursor
    DWORD cCharsWritten;
    CONSOLE_SCREEN_BUFFER_INFO csbi;
    DWORD dwConSize;

    // Get the number of character cells in the current buffer.
    if (!GetConsoleScreenBufferInfo(hConsole, &csbi))
    {
        return;
    }

    dwConSize = csbi.dwSize.X * csbi.dwSize.Y;

    // Fill the entire screen with blanks.
    if (!FillConsoleOutputCharacter(hConsole,        // Handle to console screen buffer
                                    (TCHAR)' ',      // Character to write to the buffer
                                    dwConSize,       // Number of cells to write
                                    coordScreen,     // Coordinates of first cell
                                    &cCharsWritten)) // Receive number of characters written
    {
        return;
    }

    // Get the current text attribute.
    if (!GetConsoleScreenBufferInfo(hConsole, &csbi))
    {
        return;
    }

    // Set the buffer's attributes accordingly.
    if (!FillConsoleOutputAttribute(hConsole,         // Handle to console screen buffer
                                    csbi.wAttributes, // Character attributes to use
                                    dwConSize,        // Number of cells to set attribute
                                    coordScreen,      // Coordinates of first cell
                                    &cCharsWritten))  // Receive number of characters written
    {
        return;
    }

    // Put the cursor at its home coordinates.
    SetConsoleCursorPosition(hConsole, coordScreen);
}

int main(void)
{
    HANDLE hStdout;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

    cls(hStdout);

    return 0;
}
```

Scrolling the Screen Buffer

5/18/2021 • 2 minutes to read • Edit Online

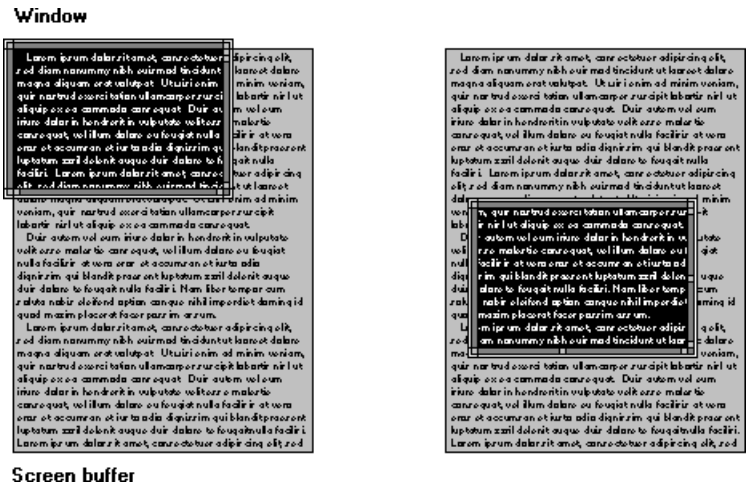
IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

The console window displays a portion of the active screen buffer. Each screen buffer maintains its own current window rectangle that specifies the coordinates of the upper left and lower right character cells to be displayed in the console window. To determine the current window rectangle of a screen buffer, use **GetConsoleScreenBufferInfo**. When a screen buffer is created, the upper left corner of its window is at the upper left corner of the console screen buffer at (0,0).

The window rectangle can change to display different parts of the console screen buffer. The window rectangle of a screen buffer can change in the following situations:

- When **SetConsoleWindowInfo** is called to specify a new window rectangle, it scrolls the view of the console screen buffer by changing the position of the window rectangle without changing the size of the window. For examples of scrolling the window's contents, see Scrolling a Screen Buffer's Window.



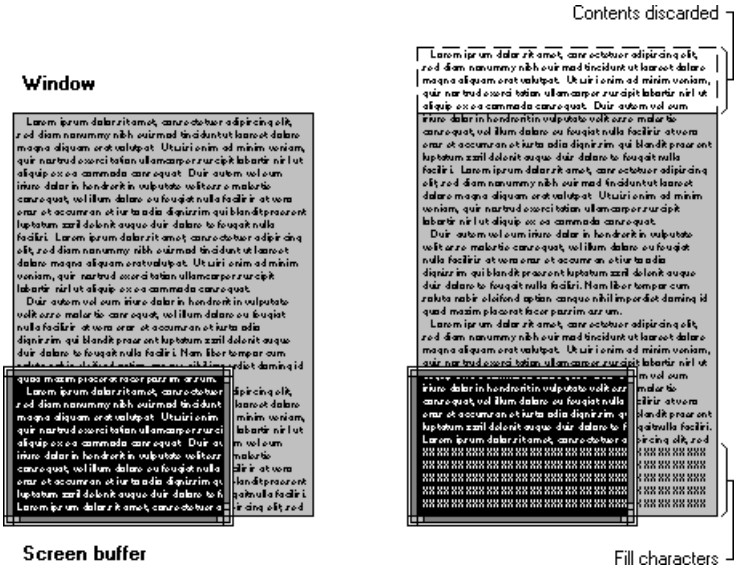
- When using the **WriteFile** function to write to a screen buffer with wrap at end-of-line (EOL) output mode enabled, the window rectangle shifts automatically, so the cursor is always displayed.
- When the **SetConsoleCursorPosition** function specifies a new cursor position that is outside the boundaries of the current window rectangle, the window rectangle shifts automatically to display the cursor.
- When the user changes the size of the console window or uses the window's scroll bars, the window rectangle of the active screen buffer can change. This change is not reported as a window resizing event in the input buffer.

In each of these situations, the window rectangle shifts to display a different part of the console screen buffer, but the contents of the console screen buffer remain in the same position. The following situations can cause the console screen buffer's contents to shift:

- When the **ScrollConsoleScreenBuffer** function is called, a rectangular block is copied from one part of a screen buffer to another.
- When using **WriteFile** to write to a screen buffer with wrap at EOL output mode enabled, the console screen buffer's contents scroll automatically when the end of the console screen buffer is encountered. This scrolling discards the top row of the console screen buffer.

ScrollConsoleScreenBuffer specifies the console screen buffer rectangle that is moved and the new upper left coordinates to which the rectangle is copied. This function can scroll a portion or the entire contents of the console screen buffer.

The illustration shows a **ScrollConsoleScreenBuffer** operation that scrolls the entire contents of the console screen buffer up by several rows. The contents of the top rows are discarded, and the bottom rows are filled with a specified character and color.



The effects of **ScrollConsoleScreenBuffer** can be limited by specifying an optional clipping rectangle so that the contents of the console screen buffer outside the clipping rectangle are unchanged. The effect of clipping is to create a subwindow (the clipping rectangle) whose contents are scrolled without affecting the rest of the console screen buffer. For an example that uses **ScrollConsoleScreenBuffer**, see Scrolling a Screen Buffer's Contents.

Scrolling a Screen Buffer's Contents

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

The ScrollConsoleScreenBuffer function moves a block of character cells from one part of a screen buffer to another part of the same screen buffer. The function specifies the upper left and lower right cells of the source rectangle to be moved and the destination coordinates of the new location for the upper left cell. The character and color data in the source cells is moved to the new location, and any cells left empty by the move are filled in with a specified character and color. If a clipping rectangle is specified, the cells outside of it are left unchanged.

ScrollConsoleScreenBuffer can be used to delete a line by specifying coordinates of the first cell in the line as the destination coordinates and specifying a scrolling rectangle that includes all the rows below the line.

The following example shows the use of a clipping rectangle to scroll only the bottom 15 rows of the console screen buffer. The rows in the specified rectangle are scrolled up one line at a time, and the top row of the block is discarded. The contents of the console screen buffer outside the clipping rectangle are left unchanged.

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

int main( void )
{
    HANDLE hStdout;
    CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
    SMALL_RECT srctScrollRect, srctClipRect;
    CHAR_INFO chiFill;
    COORD coordDest;
    int i;

    printf("\nPrinting 20 lines for reference. ");
    printf("Notice that line 6 is discarded during scrolling.\n");
    for(i=0; i<=20; i++)
        printf("%d\n", i);

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

    if (hStdout == INVALID_HANDLE_VALUE)
    {
        printf("GetStdHandle failed with %d\n", GetLastError());
        return 1;
    }

    // Get the screen buffer size.

    if (!GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        printf("GetConsoleScreenBufferInfo failed %d\n", GetLastError());
        return 1;
    }
```

```
    // The scrolling rectangle is the bottom 15 rows of the
    // screen buffer.

    srctScrollRect.Top = csbiInfo.dwSize.Y - 16;
    srctScrollRect.Bottom = csbiInfo.dwSize.Y - 1;
    srctScrollRect.Left = 0;
    srctScrollRect.Right = csbiInfo.dwSize.X - 1;

    // The destination for the scroll rectangle is one row up.

    coordDest.X = 0;
    coordDest.Y = csbiInfo.dwSize.Y - 17;

    // The clipping rectangle is the same as the scrolling rectangle.
    // The destination row is left unchanged.

    srctClipRect = srctScrollRect;

    // Fill the bottom row with green blanks.

    chiFill.Attributes = BACKGROUND_GREEN | FOREGROUND_RED;
    chiFill.Char.AsciiChar = (char)' ';

    // Scroll up one line.

    if(!ScrollConsoleScreenBuffer(
        hStdout,         // screen buffer handle
        &srctScrollRect, // scrolling rectangle
        &srctClipRect,   // clipping rectangle
        coordDest,       // top left destination cell
        &chiFill))       // fill character and color
    {
        printf("ScrollConsoleScreenBuffer failed %d\n", GetLastError());
        return 1;
    }
return 0;
}
```

Related topics

Scrolling a Screen Buffer's Window

Scrolling the Screen Buffer

Scrolling a Screen Buffer's Window

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

The SetConsoleWindowInfo function can be used to scroll the contents of a screen buffer in the console window. This function can also change the window size. The function can either specify the new upper left and lower right corners of the console screen buffer's window as absolute screen buffer coordinates or specify the changes from the current window coordinates. The function fails if the specified window coordinates are outside the boundaries of the console screen buffer.

The following example scrolls the view of the console screen buffer up by modifying the window coordinates returned by the GetConsoleScreenBufferInfo function. The `ScrollByAbsoluteCoord` function demonstrates how to specify absolute coordinates, while the `ScrollByRelativeCoord` function demonstrates how to specify relative coordinates.

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

HANDLE hStdout;

int ScrollByAbsoluteCoord(int iRows)
{
    CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
    SMALL_RECT srctWindow;

    // Get the current screen buffer size and window position.

    if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        printf("GetConsoleScreenBufferInfo (%d)\n", GetLastError());
        return 0;
    }

    // Set srctWindow to the current window size and location.

    srctWindow = csbiInfo.srWindow;

    // Check whether the window is too close to the screen buffer top

    if ( srctWindow.Top >= iRows )
    {
        srctWindow.Top -= (SHORT)iRows;     // move top up
        srctWindow.Bottom -= (SHORT)iRows;  // move bottom up

        if (! SetConsoleWindowInfo(
                   hStdout,          // screen buffer handle
                   TRUE,             // absolute coordinates
                   &srctWindow))     // specifies new location
        {
```

```
            printf("SetConsoleWindowInfo (%d)\n", GetLastError());
            return 0;
        }
        return iRows;
    }
    else
    {
        printf("\nCannot scroll; the window is too close to the top.\n");
        return 0;
    }
}

int ScrollByRelativeCoord(int iRows)
{
    CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
    SMALL_RECT srctWindow;

    // Get the current screen buffer window position.

    if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        printf("GetConsoleScreenBufferInfo (%d)\n", GetLastError());
        return 0;
    }

    // Check whether the window is too close to the screen buffer top

    if (csbiInfo.srWindow.Top >= iRows)
    {
        srctWindow.Top =- (SHORT)iRows;     // move top up
        srctWindow.Bottom =- (SHORT)iRows;  // move bottom up
        srctWindow.Left = 0;         // no change
        srctWindow.Right = 0;        // no change

        if (! SetConsoleWindowInfo(
                   hStdout,          // screen buffer handle
                   FALSE,            // relative coordinates
                   &srctWindow))     // specifies new location
        {
            printf("SetConsoleWindowInfo (%d)\n", GetLastError());
            return 0;
        }
        return iRows;
    }
    else
    {
        printf("\nCannot scroll; the window is too close to the top.\n");
        return 0;
    }
}

int main( void )
{
    int i;

    printf("\nPrinting twenty lines, then scrolling up five lines.\n");
    printf("Press any key to scroll up ten lines; ");
    printf("then press another key to stop the demo.\n");
    for(i=0; i<=20; i++)
        printf("%d\n", i);

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

    if(ScrollByAbsoluteCoord(5))
        _getch();
    else return 0;

    if(ScrollByRelativeCoord(10))
         getch();
```

```
    else return 0;
}
```

Related topics

Scrolling a Screen Buffer's Contents

Scrolling the Screen Buffer

CTRL+C and CTRL+BREAK Signals

5/18/2021 • 2 minutes to read • Edit Online

The CTRL+C and CTRL+BREAK key combinations receive special handling by console processes. By default, when a console window has the keyboard focus, CTRL+C or CTRL+BREAK is treated as a signal (SIGINT or SIGBREAK) and not as keyboard input. By default, these signals are passed to all console processes that are attached to the console. (Detached processes are not affected. See **Creation of a Console**.) The system creates a new thread in each client process to handle the event. The thread raises an exception if the process is being debugged. The debugger can handle the exception or continue with the exception unhandled.

CTRL+BREAK is always treated as a signal, but an application can change the default CTRL+C behavior in two ways that prevent the handler functions from being called:

- The **SetConsoleMode** function can disable the **ENABLE_PROCESSED_INPUT** input mode for a console's input buffer, so CTRL+C is reported as keyboard input rather than as a signal.
- When **SetConsoleCtrlHandler** is called with **NULL** and **TRUE** values for its parameters, the calling process ignores CTRL+C signals. Normal CTRL+C processing is restored by calling **SetConsoleCtrlHandler** with **NULL** and **FALSE** values. This attribute of ignoring or not ignoring CTRL+C signals is inherited by child processes, but it can be enabled or disabled by any process without affecting existing processes.

For more information on how these signals are processed, including timeouts, please see the **Handler Routine** callback documentation.

CTRL+CLOSE Signal

5/18/2021 • 2 minutes to read • Edit Online

The system generates a CTRL+CLOSE signal when the user closes a console. All processes attached to the console receive the signal, giving each process an opportunity to clean up before termination. When a process receives this signal, the handler function can take one of the following actions after performing any cleanup operations:

- Call **ExitProcess** to terminate the process.
- Return **FALSE**. If none of the registered handler functions returns **TRUE**, the default handler terminates the process.
- Return **TRUE**. In this case, no other handler functions are called and the process terminates.

Registering a Control Handler Function

9/16/2021 • 3 minutes to read • Edit Online

Basic Control Handler Example

This is an example of the **SetConsoleCtrlHandler** function that is used to install a control handler.

When a CTRL+C signal is received, the control handler returns **TRUE**, indicating that it has handled the signal. Doing this prevents other control handlers from being called.

When a **CTRL_CLOSE_EVENT** signal is received, the control handler returns **TRUE** and the process terminates.

When a **CTRL_BREAK_EVENT**, **CTRL_LOGOFF_EVENT**, or **CTRL_SHUTDOWN_EVENT** signal is received, the control handler returns **FALSE**. Doing this causes the signal to be passed to the next control handler function. If no other control handlers have been registered or none of the registered handlers returns **TRUE**, the default handler will be used, resulting in the process being terminated.

> **NOTE**
>
> Calling **AttachConsole**, **AllocConsole**, or **FreeConsole** will reset the table of control handlers in the client process to its initial state. Handlers must be registered again when the attached console session changes.

```
// CtrlHandler.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

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

BOOL WINAPI CtrlHandler(DWORD fdwCtrlType)
{
    switch (fdwCtrlType)
    {
        // Handle the CTRL-C signal.
    case CTRL_C_EVENT:
        printf("Ctrl-C event\n\n");
        Beep(750, 300);
        return TRUE;

        // CTRL-CLOSE: confirm that the user wants to exit.
    case CTRL_CLOSE_EVENT:
        Beep(600, 200);
        printf("Ctrl-Close event\n\n");
        return TRUE;

        // Pass other signals to the next handler.
    case CTRL_BREAK_EVENT:
        Beep(900, 200);
        printf("Ctrl-Break event\n\n");
        return FALSE;

    case CTRL_LOGOFF_EVENT:
        Beep(1000, 200);
        printf("Ctrl-Logoff event\n\n");
        return FALSE;

    case CTRL_SHUTDOWN_EVENT:
        Beep(750, 500);
        printf("Ctrl-Shutdown event\n\n");
        return FALSE;

    default:
        return FALSE;
    }
}

int main(void)
{
    if (SetConsoleCtrlHandler(CtrlHandler, TRUE))
    {
        printf("\nThe Control Handler is installed.\n");
        printf("\n -- Now try pressing Ctrl+C or Ctrl+Break, or");
        printf("\n    try logging off or closing the console...\n");
        printf("\n(...waiting in a loop for events...)\n\n");

        while (1) {}
    }
    else
    {
        printf("\nERROR: Could not set control handler");
        return 1;
    }
    return 0;
}
```

Listen with Hidden Window Example

Per the remarks, if the gdi32.dll or user32.dll library are loaded, **SetConsoleCtrlHandler** does not get called

for the **CTRL_LOGOFF_EVENT** and **CTRL_SHUTDOWN_EVENT** events. The workaround specified is to create a hidden window, if no window already exists, by calling the **CreateWindowEx** method with the *dwExStyle* parameter set to 0 and listen for the **WM_QUERYENDSESSION** and **WM_ENDSESSION** window messages. If a window already exists, add the two messages to the existing Window Procedure.

More information on setting up a window and its messaging loop can be found at Creating a Window.

```
// CtrlHandler.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include <Windows.h>
#include <stdio.h>

BOOL WINAPI CtrlHandler(DWORD fdwCtrlType)
{
    switch (fdwCtrlType)
    {
        // Handle the CTRL-C signal.
    case CTRL_C_EVENT:
        printf("Ctrl-C event\n\n");
        Beep(750, 300);
        return TRUE;

        // CTRL-CLOSE: confirm that the user wants to exit.
    case CTRL_CLOSE_EVENT:
        Beep(600, 200);
        printf("Ctrl-Close event\n\n");
        return TRUE;

        // Pass other signals to the next handler.
    case CTRL_BREAK_EVENT:
        Beep(900, 200);
        printf("Ctrl-Break event\n\n");
        return FALSE;

    case CTRL_LOGOFF_EVENT:
        Beep(1000, 200);
        printf("Ctrl-Logoff event\n\n");
        return FALSE;

    case CTRL_SHUTDOWN_EVENT:
        Beep(750, 500);
        printf("Ctrl-Shutdown event\n\n");
        return FALSE;

    default:
        return FALSE;
    }
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_QUERYENDSESSION:
    {
        // Check `lParam` for which system shutdown function and handle events.
        // See https://docs.microsoft.com/windows/win32/shutdown/wm-queryendsession
        return TRUE; // Respect user's intent and allow shutdown.
    }
    case WM_ENDSESSION:
    {
        // Check `lParam` for which system shutdown function and handle events.
        // See https://docs.microsoft.com/windows/win32/shutdown/wm-endsession
        return 0; // We have handled this message.
    }
    default:
```

```
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}

int main(void)
{
    WNDCLASS sampleClass{ 0 };
    sampleClass.lpszClassName = TEXT("CtrlHandlerSampleClass");
    sampleClass.lpfnWndProc = WindowProc;

    if (!RegisterClass(&sampleClass))
    {
        printf("\nERROR: Could not register window class");
        return 2;
    }

    HWND hwnd = CreateWindowEx(
        0,
        sampleClass.lpszClassName,
        TEXT("Console Control Handler Sample"),
        0,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        nullptr,
        nullptr,
        nullptr,
        nullptr
    );

    if (!hwnd)
    {
        printf("\nERROR: Could not create window");
        return 3;
    }

    ShowWindow(hwnd, SW_HIDE);

    if (SetConsoleCtrlHandler(CtrlHandler, TRUE))
    {
        printf("\nThe Control Handler is installed.\n");
        printf("\n -- Now try pressing Ctrl+C or Ctrl+Break, or");
        printf("\n    try logging off or closing the console...\n");
        printf("\n(...waiting in a loop for events...)\n\n");

        // Pump message loop for the window we created.
        MSG msg{};
        while (GetMessage(&msg, nullptr, 0, 0) > 0)
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        return 0;
    }
    else
    {
        printf("\nERROR: Could not set control handler");
        return 1;
    }
}
```

Console Virtual Terminal Sequences

9/16/2021 • 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

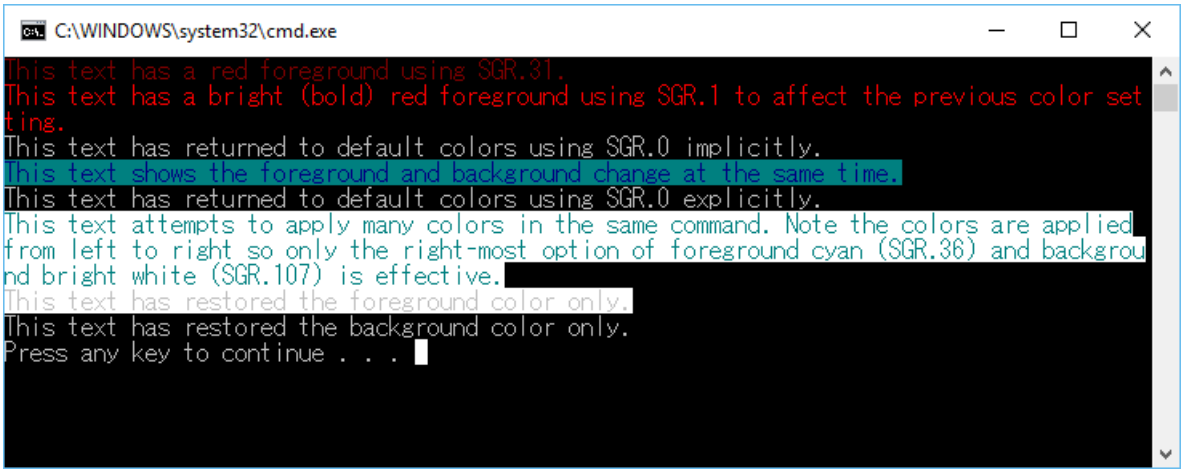
The following code provides several examples of text formatting.

```
#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");

}
```

Creating a Pseudoconsole session

8/3/2021 • 8 minutes to read • Edit Online

The Windows Pseudoconsole, sometimes also referred to as pseudo console, ConPTY, or the Windows PTY, is a mechanism designed for creating an external host for character-mode subsystem activities that replace the user interactivity portion of the default console host window.

Hosting a pseudoconsole session is a bit different than a traditional console session. Traditional console sessions automatically start when the operating system recognizes that a character-mode application is about to run. In contrast, a pseudoconsole session and the communication channels need to be created by the hosting application prior to creating the process with the child character-mode application to be hosted. The child process will still be created using the **CreateProcess** function, but with some additional information that will direct the operating system to establish the appropriate environment.

You can find additional background information about this system on the initial announcement blog post.

Complete examples of using the Pseudoconsole are available on our GitHub repository microsoft/terminal in the samples directory.

Preparing the communication channels

The first step is to create a pair of synchronous communication channels that will be provided during creation of the pseudoconsole session for bidirectional communication with the hosted application. These channels are processed by the pseudoconsole system using **ReadFile** and **WriteFile** with synchronous I/O. File or I/O device handles like a file stream or pipe are acceptable as long as an **OVERLAPPED** structure is not required for asynchronous communication.

> **WARNING**
> To prevent race conditions and deadlocks, we highly recommend that each of the communication channels is serviced on a separate thread that maintains its own client buffer state and messaging queue inside your application. Servicing all of the pseudoconsole activities on the same thread may result in a deadlock where one of the communications buffers is filled and waiting for your action while you attempt to dispatch a blocking request on another channel.

Creating the Pseudoconsole

With the communications channels that have been established, identify the "read" end of the input channel and the "write" end of the output channel. This pair of handles is provided on calling **CreatePseudoConsole** to create the object.

On creation, a size representing the X and Y dimensions (in count of characters) is required. These are the dimensions that will apply to the display surface for the final (terminal) presentation window. The values are used to create an in-memory buffer inside the pseudoconsole system.

The buffer size provide answers to client character-mode applications that probe for information using the client-side console functions like **GetConsoleScreenBufferInfoEx** and dictates the layout and positioning of text when clients use functions like **WriteConsoleOutput**.

Finally, a flags field is provided on creation of a pseudoconsole to perform special functionality. By default, set this to 0 to have no special functionality.

At this time, only one special flag is available to request the inheritence of the cursor position from a console

session already attached to the caller of the pseudoconsole API. This is intended for use in more advanced scenarios where a hosting application that is preparing a pseudoconsole session is itself also a client character-mode application of a another console environment.

A sample snippet is provided below utilizing CreatePipe to establish a pair of communication channels and create the pseudoconsole.

```
HRESULT SetUpPseudoConsole(COORD size)
{
    HRESULT hr = S_OK;

    // Create communication channels

    // - Close these after CreateProcess of child application with pseudoconsole object.
    HANDLE inputReadSide, outputWriteSide;

    // - Hold onto these and use them for communication with the child through the pseudoconsole.
    HANDLE outputReadSide, inputWriteSide;

    if (!CreatePipe(&inputReadSide, &inputWriteSide, NULL, 0))
    {
        return HRESULT_FROM_WIN32(GetLastError());
    }

    if (!CreatePipe(&outputReadSide, &outputWriteSide, NULL, 0))
    {
        return HRESULT_FROM_WIN32(GetLastError());
    }

    HPCON hPC;
    hr = CreatePseudoConsole(size, inputReadSide, outputWriteSide, 0, &hPC);
    if (FAILED(hr))
    {
        return hr;
    }

    // ...

}
```

NOTE

This snippet is incomplete and used for demonstration of this specific call only. You will need to manage the lifetime of the **HANDLE**s appropriately. Failure to manage the lifetime of **HANDLE**s correctly can result in deadlock scenarios, especially with synchronous I/O calls.

Upon completion of the CreateProcess call to create the client character-mode application attached to the pseudoconsole, the handles given during creation should be freed from this process. This will decrease the reference count on the underlying device object and allow I/O operations to properly detect a broken channel when the pseudoconsole session closes its copy of the handles.

Preparing for Creation of the Child Process

The next phase is to prepare the STARTUPINFOEX structure that will convey the pseudoconsole information while starting the child process.

This structure contains the ability to provide complex startup information including attributes for process and thread creation.

Use **InitializeProcThreadAttributeList** in a double-call fashion to first calculate the number of bytes required to hold the list, allocate the memory requested, then call again providing the opaque memory pointer to have it set up as the attribute list.

Next, call **UpdateProcThreadAttribute** passing the initialized attribute list with the flag **PROC_THREAD_ATTRIBUTE_PSEUDOCONSOLE**, the pseudoconsole handle, and the size of the pseudoconsole handle.

```
HRESULT PrepareStartupInformation(HPCON hpc, STARTUPINFOEX* psi)
{
    // Prepare Startup Information structure
    STARTUPINFOEX si;
    ZeroMemory(&si, sizeof(si));
    si.StartupInfo.cb = sizeof(STARTUPINFOEX);

    // Discover the size required for the list
    size_t bytesRequired;
    InitializeProcThreadAttributeList(NULL, 1, 0, &bytesRequired);

    // Allocate memory to represent the list
    si.lpAttributeList = (PPROC_THREAD_ATTRIBUTE_LIST)HeapAlloc(GetProcessHeap(), 0, bytesRequired);
    if (!si.lpAttributeList)
    {
        return E_OUTOFMEMORY;
    }

    // Initialize the list memory location
    if (!InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0, &bytesRequired))
    {
        HeapFree(GetProcessHeap(), 0, si.lpAttributeList);
        return HRESULT_FROM_WIN32(GetLastError());
    }

    // Set the pseudoconsole information into the list
    if (!UpdateProcThreadAttribute(si.lpAttributeList,
                                   0,
                                   PROC_THREAD_ATTRIBUTE_PSEUDOCONSOLE,
                                   hpc,
                                   sizeof(hpc),
                                   NULL,
                                   NULL))
    {
        HeapFree(GetProcessHeap(), 0, si.lpAttributeList);
        return HRESULT_FROM_WIN32(GetLastError());
    }

    *psi = si;

    return S_OK;
}
```

Creating the Hosted Process

Next, call **CreateProcess** passing the **STARTUPINFOEX** structure along with the path to the executable and any additional configuration information if applicable. It is important to set the **EXTENDED_STARTUPINFO_PRESENT** flag when calling to alert the system that the pseudoconsole reference is contained in the extended information.

```
HRESULT SetUpPseudoConsole(COORD size)
{
    // ...

    PCWSTR childApplication = L"C:\\windows\\system32\\cmd.exe";

    // Create mutable text string for CreateProcessW command line string.
    const size_t charsRequired = wcslen(childApplication) + 1; // +1 null terminator
    PWSTR cmdLineMutable = (PWSTR)HeapAlloc(GetProcessHeap(), 0, sizeof(wchar_t) * charsRequired);

    if (!cmdLineMutable)
    {
        return E_OUTOFMEMORY;
    }

    wcscpy_s(cmdLineMutable, charsRequired, childApplication);

    PROCESS_INFORMATION pi;
    ZeroMemory(&pi, sizeof(pi));

    // Call CreateProcess
    if (!CreateProcessW(NULL,
                        cmdLineMutable,
                        NULL,
                        NULL,
                        FALSE,
                        EXTENDED_STARTUPINFO_PRESENT,
                        NULL,
                        NULL,
                        &siEx.StartupInfo,
                        &pi))
    {
        HeapFree(GetProcessHeap(), 0, cmdLineMutable);
        return HRESULT_FROM_WIN32(GetLastError());
    }

    // ...
}
```

NOTE

Closing the pseudoconsole session while the hosted process is still starting up and connecting can result in an error dialog being shown by the client application. The same error dialog is shown if the hosted process is given an invalid pseudoconsole handle for startup. To the hosted process initialization code, the two circumstances are identical. The pop-up dialog from the hosted client application on failure will read `0xc0000142` with a localized message detailing failure to initialize.

Communicating with the Pseudoconsole Session

Once the process is created successfully, the hosting application can use the write end of the input pipe to send user interaction information into the pseudoconsole and the read end of the output pipe to receive graphical presentation information from the pseudo console.

It is completely up to the hosting application to decide how to handle further activity. The hosting application could launch a window in another thread to collect user interaction input and serialize it into the write end of the input pipe for the pseudoconsole and the hosted character-mode application. Another thread could be launched to drain the read end of the output pipe for the pseudoconsole, decode the text and virtual terminal sequence information, and present that to the screen.

Threads could also be used to relay the information from the pseudoconsole channels out to a different channel

or device including a network to remote information to another process or machine and avoiding any local transcoding of the information.

Resizing the Pseudoconsole

Throughout the course of runtime, there may be a circumstance by which the size of the buffer needs to be changed due to a user interaction or a request received out of band from another display/interaction device.

This can be done with the **ResizePseudoConsole** function specifying both the height and width of the buffer in a count of characters.

```
// Theoretical event handler function with theoretical
// event that has associated display properties
// on Source property.
void OnWindowResize(Event e)
{
    // Retrieve width and height dimensions of display in
    // characters using theoretical height/width functions
    // that can retrieve the properties from the display
    // attached to the event.
    COORD size;
    size.X = GetViewWidth(e.Source);
    size.Y = GetViewHeight(e.Source);

    // Call pseudoconsole API to inform buffer dimension update
    ResizePseudoConsole(m_hpc, size);
}
```

Ending the Pseudoconsole Session

To end the session, call the **ClosePseudoConsole** function with the handle from the original pseudoconsole creation. Any attached client character-mode applications, such as the one from the **CreateProcess** call, will be terminated when the session is closed. If the original child was a shell-type application that creates other processes, any related attached processes in the tree will also be terminated.

WARNING

Closing the session has several side effects which can result in a deadlock condition if the pseudoconsole is used in a single-threaded synchronous fashion. The act of closing the pseudoconsole session may emit a final frame update to `hOutput` which should be drained from the communications channel buffer. Additionally, if `PSEUDOCONSOLE_INHERIT_CURSOR` was selected while creating the pseudoconsole, attempting to close the pseudoconsole without responding to the cursor inheritence query message (received on `hOutput` and replied to via `hInput`) may result in another deadlock condition. It is recommended that communications channels for the pseudoconsole are serviced on individual threads and remain drained and processed until broken of their own accord by the client application exiting or by the completion of teardown activities in calling the **ClosePseudoConsole** function.

Console Functions

5/18/2021 • 4 minutes to read • Edit Online

The following functions are used to access a console.

FUNCTION	DESCRIPTION
AddConsoleAlias	Defines a console alias for the specified executable.
AllocConsole	Allocates a new console for the calling process.
AttachConsole	Attaches the calling process to the console of the specified process.
ClosePseudoConsole	Closes a pseudoconsole from the given handle.
CreatePseudoConsole	Allocates a new pseudoconsole for the calling process.
CreateConsoleScreenBuffer	Creates a console screen buffer.
FillConsoleOutputAttribute	Sets the text and background color attributes for a specified number of character cells.
FillConsoleOutputCharacter	Writes a character to the console screen buffer a specified number of times.
FlushConsoleInputBuffer	Flushes the console input buffer.
FreeConsole	Detaches the calling process from its console.
GenerateConsoleCtrlEvent	Sends a specified signal to a console process group that shares the console associated with the calling process.
GetConsoleAlias	Retrieves the specified alias for the specified executable.
GetConsoleAliases	Retrieves all defined console aliases for the specified executable.
GetConsoleAliasesLength	Returns the size, in bytes, of the buffer needed to store all of the console aliases for the specified executable.
GetConsoleAliasExes	Retrieves the names of all executables with console aliases defined.
GetConsoleAliasExesLength	Returns the size, in bytes, of the buffer needed to store the names of all executables that have console aliases defined.
GetConsoleCP	Retrieves the input code page used by the console associated with the calling process.

FUNCTION	DESCRIPTION
GetConsoleCursorInfo	Retrieves information about the size and visibility of the cursor for the specified console screen buffer.
GetConsoleDisplayMode	Retrieves the display mode of the current console.
GetConsoleFontSize	Retrieves the size of the font used by the specified console screen buffer.
GetConsoleHistoryInfo	Retrieves the history settings for the calling process's console.
GetConsoleMode	Retrieves the current input mode of a console's input buffer or the current output mode of a console screen buffer.
GetConsoleOriginalTitle	Retrieves the original title for the current console window.
GetConsoleOutputCP	Retrieves the output code page used by the console associated with the calling process.
GetConsoleProcessList	Retrieves a list of the processes attached to the current console.
GetConsoleScreenBufferInfo	Retrieves information about the specified console screen buffer.
GetConsoleScreenBufferInfoEx	Retrieves extended information about the specified console screen buffer.
GetConsoleSelectionInfo	Retrieves information about the current console selection.
GetConsoleTitle	Retrieves the title for the current console window.
GetConsoleWindow	Retrieves the window handle used by the console associated with the calling process.
GetCurrentConsoleFont	Retrieves information about the current console font.
GetCurrentConsoleFontEx	Retrieves extended information about the current console font.
GetLargestConsoleWindowSize	Retrieves the size of the largest possible console window.
GetNumberOfConsoleInputEvents	Retrieves the number of unread input records in the console's input buffer.
GetNumberOfConsoleMouseButtons	Retrieves the number of buttons on the mouse used by the current console.
GetStdHandle	Retrieves a handle for the standard input, standard output, or standard error device.
HandlerRoutine	An application-defined function used with the SetConsoleCtrlHandler function.

FUNCTION	DESCRIPTION
PeekConsoleInput	Reads data from the specified console input buffer without removing it from the buffer.
ReadConsole	Reads character input from the console input buffer and removes it from the buffer.
ReadConsoleInput	Reads data from a console input buffer and removes it from the buffer.
ReadConsoleOutput	Reads character and color attribute data from a rectangular block of character cells in a console screen buffer.
ReadConsoleOutputAttribute	Copies a specified number of foreground and background color attributes from consecutive cells of a console screen buffer.
ReadConsoleOutputCharacter	Copies a number of characters from consecutive cells of a console screen buffer.
ResizePseudoConsole	Resizes the internal buffers for a pseudoconsole to the given size.
ScrollConsoleScreenBuffer	Moves a block of data in a screen buffer.
SetConsoleActiveScreenBuffer	Sets the specified screen buffer to be the currently displayed console screen buffer.
SetConsoleCP	Sets the input code page used by the console associated with the calling process.
SetConsoleCtrlHandler	Adds or removes an application-defined HandlerRoutine from the list of handler functions for the calling process.
SetConsoleCursorInfo	Sets the size and visibility of the cursor for the specified console screen buffer.
SetConsoleCursorPosition	Sets the cursor position in the specified console screen buffer.
SetConsoleDisplayMode	Sets the display mode of the specified console screen buffer.
SetConsoleHistoryInfo	Sets the history settings for the calling process's console.
SetConsoleMode	Sets the input mode of a console's input buffer or the output mode of a console screen buffer.
SetConsoleOutputCP	Sets the output code page used by the console associated with the calling process.
SetConsoleScreenBufferInfoEx	Sets extended information about the specified console screen buffer.
SetConsoleScreenBufferSize	Changes the size of the specified console screen buffer.

FUNCTION	DESCRIPTION
SetConsoleTextAttribute	Sets the foreground (text) and background color attributes of characters written to the console screen buffer.
SetConsoleTitle	Sets the title for the current console window.
SetConsoleWindowInfo	Sets the current size and position of a console screen buffer's window.
SetCurrentConsoleFontEx	Sets extended information about the current console font.
SetStdHandle	Sets the handle for the standard input, standard output, or standard error device.
WriteConsole	Writes a character string to a console screen buffer beginning at the current cursor location.
WriteConsoleInput	Writes data directly to the console input buffer.
WriteConsoleOutput	Writes character and color attribute data to a specified rectangular block of character cells in a console screen buffer.
WriteConsoleOutputAttribute	Copies a number of foreground and background color attributes to consecutive cells of a console screen buffer.
WriteConsoleOutputCharacter	Copies a number of characters to consecutive cells of a console screen buffer.

AddConsoleAlias function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Defines a console alias for the specified executable.

Syntax

```
BOOL WINAPI AddConsoleAlias(
  _In_ LPCTSTR Source,
  _In_ LPCTSTR Target,
  _In_ LPCTSTR ExeName
);
```

Parameters

Source [in]
The console alias to be mapped to the text specified by *Target*.

Target [in]
The text to be substituted for *Source*. If this parameter is **NULL**, then the console alias is removed.

ExeName [in]
The name of the executable file for which the console alias is to be defined.

Return value

If the function succeeds, the return value is **TRUE**.

If the function fails, the return value is **FALSE**. To get extended error information, call GetLastError.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0501 or later. For more information, see Using the Windows Headers.

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application acting as a shell or interpreter is expected to maintain its own user-convenience functionality like line reading and manipulation behavior including aliases and command history. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Examples

For an example, see Console Aliases.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**AddConsoleAliasW** (Unicode) and **AddConsoleAliasA** (ANSI)

See also

Console Aliases

Console Functions

GetConsoleAlias

GetConsoleAliases

GetConsoleAliasExes

AllocConsole function

5/18/2021 • 2 minutes to read • Edit Online

Allocates a new console for the calling process.

Syntax

```
BOOL WINAPI AllocConsole(void);
```

Parameters

This function has no parameters.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A process can be associated with only one console, so the **AllocConsole** function fails if the calling process already has a console. A process can use the **FreeConsole** function to detach itself from its current console, then it can call **AllocConsole** to create a new console or **AttachConsole** to attach to another console.

If the calling process creates a child process, the child inherits the new console.

AllocConsole initializes standard input, standard output, and standard error handles for the new console. The standard input handle is a handle to the console's input buffer, and the standard output and standard error handles are handles to the console's screen buffer. To retrieve these handles, use the **GetStdHandle** function.

This function is primarily used by a graphical user interface (GUI) application to create a console window. GUI applications are initialized without a console. Console applications are initialized with a console, unless they are created as detached processes (by calling the **CreateProcess** function with the **DETACHED_PROCESS** flag).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Consoles

AttachConsole

CreateProcess

FreeConsole

GetStdHandle

AttachConsole function

9/16/2021 • 2 minutes to read • Edit Online

Attaches the calling process to the console of the specified process as a client application.

Syntax

```
BOOL WINAPI AttachConsole(
  _In_ DWORD dwProcessId
);
```

Parameters

dwProcessId [in]
The identifier of the process whose console is to be used. This parameter can be one of the following values.

VALUE	MEANING
pid	Use the console of the specified process.
ATTACH_PARENT_PROCESS `(DWORD)-1`	Use the console of the parent of the current process.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A process can be attached to at most one console. If the calling process is already attached to a console, the error code returned is **ERROR_ACCESS_DENIED**. If the specified process does not have a console, the error code returned is **ERROR_INVALID_HANDLE**. If the specified process does not exist, the error code returned is **ERROR_INVALID_PARAMETER**.

A process can use the **FreeConsole** function to detach itself from its console. If other processes share the console, the console is not destroyed, but the process that called **FreeConsole** cannot refer to it. A console is closed when the last process attached to it terminates or calls **FreeConsole**. After a process calls **FreeConsole**, it can call the **AllocConsole** function to create a new console or **AttachConsole** to attach to another console.

This function is primarily useful to applications that were linked with **/SUBSYSTEM:WINDOWS**, which implies to the operating system that a console is not needed before entering the program's main method. In that instance, the standard handles retrieved with **GetStdHandle** will likely be invalid on startup until **AttachConsole** is called. The exception to this is if the application is launched with handle inheritance by its parent process.

To compile an application that uses this function, define **_WIN32_WINNT** as `0x0501` or later. For more information, see Using the Windows Headers.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Consoles

AllocConsole

FreeConsole

GetConsoleProcessList

ClosePseudoConsole function

5/18/2021 • 2 minutes to read • Edit Online

Closes a pseudoconsole from the given handle.

Syntax

```
void WINAPI ClosePseudoConsole(
    _In_ HPCON hPC
);
```

Parameters

hPC [in]
A handle to an active pseudoconsole as opened by CreatePseudoConsole.

Return value

none

Remarks

Upon closing a pseudoconsole, client applications attached to the session will be terminated as well.

A final painted frame may arrive on the `hOutput` handle originally provided to CreatePsuedoConsole when this API is called. It is expected that the caller will drain this information from the communication channel buffer and either present it or discard it. Failure to drain the buffer may cause the Close call to wait indefinitely until it is drained or the communication channels are broken another way.

Requirements

Minimum supported client	Windows 10 October 2018 Update (version 1809) [desktop apps only]
Minimum supported server	Windows Server 2019 [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Pseudoconsoles

CreatePseudoConsole

CreateConsoleScreenBuffer function

5/18/2021 • 3 minutes to read • Edit Online

> **IMPORTANT**
>
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Creates a console screen buffer.

Syntax

```
HANDLE WINAPI CreateConsoleScreenBuffer(
  _In_             DWORD                dwDesiredAccess,
  _In_             DWORD                dwShareMode,
  _In_opt_   const SECURITY_ATTRIBUTES *lpSecurityAttributes,
  _In_             DWORD                dwFlags,
  _Reserved_       LPVOID               lpScreenBufferData
);
```

Parameters

dwDesiredAccess [in]
The access to the console screen buffer. For a list of access rights, see Console Buffer Security and Access Rights.

dwShareMode [in]
This parameter can be zero, indicating that the buffer cannot be shared, or it can be one or more of the following values.

VALUE	MEANING
FILE_SHARE_READ 0x00000001	Other open operations can be performed on the console screen buffer for read access.
FILE_SHARE_WRITE 0x00000002	Other open operations can be performed on the console screen buffer for write access.

lpSecurityAttributes [in, optional]
A pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpSecurityAttributes* is **NULL**, the handle cannot be inherited. The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new console screen buffer. If *lpSecurityAttributes* is **NULL**, the console screen buffer gets a default security descriptor. The ACLs in the default security descriptor for a console screen buffer come from the primary or impersonation token of the creator.

dwFlags [in]
The type of console screen buffer to create. The only supported screen buffer type is **CONSOLE_TEXTMODE_BUFFER**.

lpScreenBufferData
Reserved; should be **NULL**.

Return value

If the function succeeds, the return value is a handle to the new console screen buffer.

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call **GetLastError**.

Remarks

A console can have multiple screen buffers but only one active screen buffer. Inactive screen buffers can be accessed for reading and writing, but only the active screen buffer is displayed. To make the new screen buffer the active screen buffer, use the **SetConsoleActiveScreenBuffer** function.

The newly created screen buffer will copy some properties from the active screen buffer at the time that this function is called. The behavior is as follows:

- `Font` - copied from active screen buffer
- `Display Window Size` - copied from active screen buffer
- `Buffer Size` - matched to `Display Window Size` (**NOT** copied)
- `Default Attributes` (colors) - copied from active screen buffer
- `Default Popup Attributes` (colors) - copied from active screen buffer

The calling process can use the returned handle in any function that requires a handle to a console screen buffer, subject to the limitations of access specified by the *dwDesiredAccess* parameter.

The calling process can use the **DuplicateHandle** function to create a duplicate screen buffer handle that has different access or inheritability from the original handle. However, **DuplicateHandle** cannot be used to create a duplicate that is valid for a different process (except through inheritance).

To close the console screen buffer handle, use the **CloseHandle** function.

> **TIP**
> This API is not recommended but it does have an approximate **virtual terminal** equivalent in the **alternate screen buffer** sequence. Setting the *alternate screen buffer* can provide an application with a separate, isolated space for drawing over the course of its session runtime while preserving the content that was displayed by the application's invoker. This maintains that drawing information for simple restoration on process exit.

Examples

For an example, see Reading and Writing Blocks of Characters and Attributes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Screen Buffers

CloseHandle

DuplicateHandle

GetConsoleScreenBufferInfo

SECURITY_ATTRIBUTES

SetConsoleActiveScreenBuffer

SetConsoleScreenBufferSize

CreatePseudoConsole function

5/18/2021 • 2 minutes to read • Edit Online

Creates a new pseudoconsole object for the calling process.

Syntax

```
HRESULT WINAPI CreatePseudoConsole(
    _In_ COORD size,
    _In_ HANDLE hInput,
    _In_ HANDLE hOutput,
    _In_ DWORD dwFlags,
    _Out_ HPCON* phPC
);
```

Parameters

size [in]
The dimensions of the window/buffer in count of characters that will be used on initial creation of the pseudoconsole. This can be adjusted later with ResizePseudoConsole.

hInput [in]
An open handle to a stream of data that represents user input to the device. This is currently restricted to synchronous I/O.

hOutput [in]
An open handle to a stream of data that represents application output from the device. This is currently restricted to synchronous I/O.

dwFlags [in]
The value can be one of the following:

VALUE	MEANING
0	Perform a standard pseudoconsole creation.
PSEUDOCONSOLE_INHERIT_CURSOR (DWORD)1	The created pseudoconsole session will attempt to inherit the cursor position of the parent console.

phPC [out]
Pointer to a location that will receive a handle to the new pseudoconsole device.

Return value

Type: **HRESULT**

If this method succeeds, it returns **S_OK**. Otherwise, it returns an **HRESULT** error code.

Remarks

This function is primarily used by applications attempting to be a terminal window for a command-line user interface (CUI) application. The callers become responsible for presentation of the information on the output

stream and for collecting user input and serializing it into the input stream.

The input and output streams encoded as UTF-8 contain plain text interleaved with Virtual Terminal Sequences.

On the output stream, the virtual terminal sequences can be decoded by the calling application to layout and present the plain text in a display window.

On the input stream, plain text represents standard keyboard keys input by a user. More complicated operations are represented by encoding control keys and mouse movements as virtual terminal sequences embedded in this stream.

The handle created by this function must be closed with ClosePseudoConsole when operations are complete.

If using `PSEUDOCONSOLE_INHERIT_CURSOR`, the calling application should be prepared to respond to the request for the cursor state in an asynchronous fashion on a background thread by forwarding or interpreting the request for cursor information that will be received on `hOutput` and replying on `hInput`. Failure to do so may cause the calling application to hang while making another request of the pseudoconsole system.

Examples

For a full walkthrough on using this function to establish a pseudoconsole session, please see Creating a Pseudoconsole Session.

Requirements

Minimum supported client	Windows 10 October 2018 Update (version 1809) [desktop apps only]
Minimum supported server	Windows Server 2019 [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Pseudoconsoles

Creating a Pseudoconsole Session

ResizePseudoConsole

ClosePseudoConsole

FillConsoleOutputAttribute function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets the character attributes for a specified number of character cells, beginning at the specified coordinates in a screen buffer.

Syntax

```
BOOL WINAPI FillConsoleOutputAttribute(
  _In_  HANDLE  hConsoleOutput,
  _In_  WORD    wAttribute,
  _In_  DWORD   nLength,
  _In_  COORD   dwWriteCoord,
  _Out_ LPDWORD lpNumberOfAttrsWritten
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

wAttribute [in]
The attributes to use when writing to the console screen buffer. For more information, see Character Attributes.

nLength [in]
The number of character cells to be set to the specified color attributes.

dwWriteCoord [in]
A **COORD** structure that specifies the character coordinates of the first cell whose attributes are to be set.

lpNumberOfAttrsWritten [out]
A pointer to a variable that receives the number of character cells whose attributes were actually set.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the number of character cells whose attributes are to be set extends beyond the end of the specified row in the

console screen buffer, the cells of the next row are set. If the number of cells to write to extends beyond the end of the console screen buffer, the cells are written up to the end of the console screen buffer.

The character values at the positions written to are not changed.

> **TIP**
> This API is not recommended and does not have a specific **virtual terminal** equivalent. Filling the region outside the viewable window is not supported and is reserved for the terminal's history space. Filling a visible region with new text or color is performed through **moving the cursor**, **setting the new attributes**, then writing the desired text for that region, repeating characters if necessary for the length of the fill run. Additional cursor movement may be required followed by writing the desired text to fill a rectangular region. The client application is expected to keep its own memory of what is on the screen and is not able to query the remote state. More information can be found in **classic console versus virtual terminal** documentation.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

COORD

FillConsoleOutputCharacter

Low-Level Console Output Functions

SetConsoleTextAttribute

WriteConsoleOutputAttribute

FillConsoleOutputCharacter function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Writes a character to the console screen buffer a specified number of times, beginning at the specified coordinates.

Syntax

```
BOOL WINAPI FillConsoleOutputCharacter(
  _In_  HANDLE  hConsoleOutput,
  _In_  TCHAR   cCharacter,
  _In_  DWORD   nLength,
  _In_  COORD   dwWriteCoord,
  _Out_ LPDWORD lpNumberOfCharsWritten
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

cCharacter [in]
The character to be written to the console screen buffer.

nLength [in]
The number of character cells to which the character should be written.

dwWriteCoord [in]
A **COORD** structure that specifies the character coordinates of the first cell to which the character is to be written.

lpNumberOfCharsWritten [out]
A pointer to a variable that receives the number of characters actually written to the console screen buffer.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the number of characters to write to extends beyond the end of the specified row in the console screen buffer, characters are written to the next row. If the number of characters to write to extends beyond the end of the console screen buffer, the characters are written up to the end of the console screen buffer.

The attribute values at the positions written are not changed.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the SetConsoleCP or SetConsoleOutputCP functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

> **TIP**
> This API is not recommended and does not have a specific **virtual terminal** equivalent. Filling the region outside the viewable window is not supported and is reserved for the terminal's history space. Filling a visible region with new text or color is performed through **moving the cursor**, **setting the new attributes**, then writing the desired text for that region, repeating characters if necessary for the length of the fill run. Additional cursor movement may be required followed by writing the desired text to fill a rectangular region. The client application is expected to keep its own memory of what is on the screen and is not able to query the remote state. More information can be found in **classic console versus virtual terminal** documentation.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**FillConsoleOutputCharacterW** (Unicode) and **FillConsoleOutputCharacterA** (ANSI)

See also

Console Functions

COORD

FillConsoleOutputAttribute

Low-Level Console Output Functions

SetConsoleCP

SetConsoleOutputCP

WriteConsoleOutputCharacter

FlushConsoleInputBuffer function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Flushes the console input buffer. All input records currently in the input buffer are discarded.

Syntax

```
BOOL WINAPI FlushConsoleInputBuffer(
  _In_ HANDLE hConsoleInput
);
```

Parameters

hConsoleInput [in]
A handle to the console input buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. Attempting to empty the input queue all at once can destroy state in the queue in an unexpected manner.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Low-Level Console Input Functions

GetNumberOfConsoleInputEvents

PeekConsoleInput

ReadConsoleInput

WriteConsoleInput

FreeConsole function

5/18/2021 • 2 minutes to read • Edit Online

Detaches the calling process from its console.

Syntax

```
BOOL WINAPI FreeConsole(void);
```

Parameters

This function has no parameters.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A process can be attached to at most one console. If the calling process is not already attached to a console, the error code returned is **ERROR_INVALID_PARAMETER** (87).

A process can use the **FreeConsole** function to detach itself from its console. If other processes share the console, the console is not destroyed, but the process that called **FreeConsole** cannot refer to it. A console is closed when the last process attached to it terminates or calls **FreeConsole**. After a process calls **FreeConsole**, it can call the **AllocConsole** function to create a new console or **AttachConsole** to attach to another console.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

AllocConsole

AttachConsole

Console Functions

Consoles

GenerateConsoleCtrlEvent function

9/16/2021 • 2 minutes to read • Edit Online

Sends a specified signal to a console process group that shares the console associated with the calling process.

Syntax

```
BOOL WINAPI GenerateConsoleCtrlEvent(
  _In_ DWORD dwCtrlEvent,
  _In_ DWORD dwProcessGroupId
);
```

Parameters

dwCtrlEvent [in]
The type of signal to be generated. This parameter can be one of the following values.

VALUE	MEANING
CTRL_C_EVENT 0	Generates a CTRL+C signal. This signal cannot be limited to a specific process group. If *dwProcessGroupId* is nonzero, this function will succeed, but the CTRL+C signal will not be received by processes within the specified process group.
CTRL_BREAK_EVENT 1	Generates a CTRL+BREAK signal.

dwProcessGroupId [in]
The identifier of the process group to receive the signal. A process group is created when the **CREATE_NEW_PROCESS_GROUP** flag is specified in a call to the **CreateProcess** function. The process identifier of the new process is also the process group identifier of a new process group. The process group includes all processes that are descendants of the root process. Only those processes in the group that share the same console as the calling process receive the signal. In other words, if a process in the group creates a new console, that process does not receive the signal, nor do its descendants.

If this parameter is zero, the signal is generated in all processes that share the console of the calling process.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

GenerateConsoleCtrlEvent causes the control handler functions of processes in the target group to be called. All console processes have a default handler function that calls the **ExitProcess** function. A console process can use the **SetConsoleCtrlHandler** function to install or remove other handler functions.

SetConsoleCtrlHandler can also enable an inheritable attribute that causes the calling process to ignore CTRL+C signals. If **GenerateConsoleCtrlEvent** sends a CTRL+C signal to a process for which this attribute is enabled, the handler functions for that process are not called. CTRL+BREAK signals always cause the handler

functions to be called.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Control Handlers

Console Functions

CreateProcess

ExitProcess

SetConsoleCtrlHandler

GetConsoleAlias function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the text for the specified console alias and executable.

Syntax

```
DWORD WINAPI GetConsoleAlias(
  _In_  LPTSTR lpSource,
  _Out_ LPTSTR lpTargetBuffer,
  _In_  DWORD  TargetBufferLength,
  _In_  LPTSTR lpExeName
);
```

Parameters

lpSource [in]
The console alias whose text is to be retrieved.

lpTargetBuffer [out]
A pointer to a buffer that receives the text associated with the console alias.

TargetBufferLength [in]
The size of the buffer pointed to by *lpTargetBuffer*, in bytes.

lpExeName [in]
The name of the executable file.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0501 or later. For more information, see Using the Windows Headers.

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application acting as a shell or interpreter is expected to maintain its own user-convenience functionality like line reading and manipulation behavior including aliases and command history. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**GetConsoleAliasW** (Unicode) and **GetConsoleAliasA** (ANSI)

See also

Console Aliases

Console Functions

AddConsoleAlias

GetConsoleAliases

GetConsoleAliasExes

GetConsoleAliases function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves all defined console aliases for the specified executable.

Syntax

```
DWORD WINAPI GetConsoleAliases(
  _Out_ LPTSTR lpAliasBuffer,
  _In_  DWORD  AliasBufferLength,
  _In_  LPTSTR lpExeName
);
```

Parameters

lpAliasBuffer [out]
A pointer to a buffer that receives the aliases.

The format of the data is as follows: *Source1*=*Target1*\0*Source2*=*Target2*\0... *SourceN*=*TargetN*\0, where *N* is the number of console aliases defined.

AliasBufferLength [in]
The size of the buffer pointed to by *lpAliasBuffer*, in bytes.

lpExeName [in]
The executable file whose aliases are to be retrieved.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

To determine the required size for the *lpExeName* buffer, use the **GetConsoleAliasesLength** function.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0501 or later. For more information, see Using the Windows Headers.

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application acting as a shell or interpreter is expected to maintain its own user-convenience functionality like line reading and manipulation behavior including aliases and command history. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**GetConsoleAliasesW** (Unicode) and **GetConsoleAliasesA** (ANSI)

See also

AddConsoleAlias

Console Aliases

Console Functions

GetConsoleAlias

GetConsoleAliasesLength

GetConsoleAliasExes

GetConsoleAliasesLength function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the required size for the buffer used by the **GetConsoleAliases** function.

Syntax

```
DWORD WINAPI GetConsoleAliasesLength(
  _In_ LPTSTR lpExeName
);
```

Parameters

lpExeName [in]
The name of the executable file whose console aliases are to be retrieved.

Return value

The size of the buffer required to store all console aliases defined for this executable file, in bytes.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0501 or later. For more information, see Using the Windows Headers.

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application acting as a shell or interpreter is expected to maintain its own user-convenience functionality like line reading and manipulation behavior including aliases and command history. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**GetConsoleAliasesLengthW** (Unicode) and **GetConsoleAliasesLengthA** (ANSI)

See also

AddConsoleAlias

Console Aliases

Console Functions

GetConsoleAlias

GetConsoleAliases

GetConsoleAliasExes

GetConsoleAliasExes function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the names of all executable files with console aliases defined.

Syntax

```
DWORD WINAPI GetConsoleAliasExes(
  _Out_ LPTSTR lpExeNameBuffer,
  _In_  DWORD  ExeNameBufferLength
);
```

Parameters

lpExeNameBuffer [out]
A pointer to a buffer that receives the names of the executable files.

ExeNameBufferLength [in]
The size of the buffer pointed to by *lpExeNameBuffer*, in bytes.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

To determine the required size for the *lpExeNameBuffer* buffer, use the **GetConsoleAliasExesLength** function.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0501 or later. For more information, see Using the Windows Headers.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application acting as a shell or interpreter is expected to maintain its own user-convenience functionality like line reading and manipulation behavior including aliases and command history. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**GetConsoleAliasExesW** (Unicode) and **GetConsoleAliasExesA** (ANSI)

See also

AddConsoleAlias

Console Aliases

Console Functions

GetConsoleAlias

GetConsoleAliasExesLength

GetConsoleAliases

GetConsoleAliasExesLength function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
>
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the required size for the buffer used by the GetConsoleAliasExes function.

Syntax

```
DWORD WINAPI GetConsoleAliasExesLength(void);
```

Parameters

This function has no parameters.

Return value

The size of the buffer required to store the names of all executable files that have console aliases defined, in bytes.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0501 or later. For more information, see Using the Windows Headers.

> **TIP**
>
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application acting as a shell or interpreter is expected to maintain its own user-convenience functionality like line reading and manipulation behavior including aliases and command history. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**GetConsoleAliasExesLengthW** (Unicode) and **GetConsoleAliasExesLengthA** (ANSI)

See also

AddConsoleAlias

Console Aliases

Console Functions

GetConsoleAlias

GetConsoleAliases

GetConsoleAliasExes

GetConsoleCP function

5/18/2021 • 2 minutes to read • Edit Online

Retrieves the input code page used by the console associated with the calling process. A console uses its input code page to translate keyboard input into the corresponding character value.

Syntax

```
UINT WINAPI GetConsoleCP(void);
```

Parameters

This function has no parameters.

Return value

The return value is a code that identifies the code page. For a list of identifiers, see Code Page Identifiers.

If the return value is zero, the function has failed. To get extended error information, call **GetLastError**.

Remarks

A code page maps 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages. To retrieve more information about a code page, including it's name, see the **GetCPInfoEx** function.

To set a console's input code page, use the **SetConsoleCP** function. To set and query a console's output code page, use the **SetConsoleOutputCP** and **GetConsoleOutputCP** functions.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Code Pages

Console Functions

GetConsoleOutputCP

SetConsoleCP

SetConsoleOutputCP

GetConsoleCursorInfo function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves information about the size and visibility of the cursor for the specified console screen buffer.

Syntax

```
BOOL WINAPI GetConsoleCursorInfo(
  _In_  HANDLE               hConsoleOutput,
  _Out_ PCONSOLE_CURSOR_INFO lpConsoleCursorInfo
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpConsoleCursorInfo [out]
A pointer to a **CONSOLE_CURSOR_INFO** structure that receives information about the console's cursor.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Screen Buffers

CONSOLE_CURSOR_INFO

SetConsoleCursorInfo

GetConsoleDisplayMode function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the display mode of the current console.

Syntax

```
BOOL WINAPI GetConsoleDisplayMode(
  _Out_ LPDWORD lpModeFlags
);
```

Parameters

lpModeFlags [out]
The display mode of the console. This parameter can be one or more of the following values.

VALUE	MEANING
CONSOLE_FULLSCREEN 1	Full-screen console. The console is in this mode as soon as the window is maximized. At this point, the transition to full-screen mode can still fail.
CONSOLE_FULLSCREEN_HARDWARE 2	Full-screen console communicating directly with the video hardware. This mode is set after the console is in **CONSOLE_FULLSCREEN** mode to indicate that the transition to full-screen mode has completed.

NOTE

The transition to a 100% full screen video hardware mode was removed in Windows Vista with the replatforming of the graphics stack to WDDM. On later versions of Windows, the maximum resulting state is **CONSOLE_FULLSCREEN** representing a frameless window that appears full screen but isn't in exclusive control of the hardware.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Modes

SetConsoleDisplayMode

GetConsoleFontSize function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the size of the font used by the specified console screen buffer.

Syntax

```
COORD WINAPI GetConsoleFontSize(
  _In_ HANDLE hConsoleOutput,
  _In_ DWORD  nFont
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

nFont [in]
The index of the font whose size is to be retrieved. This index is obtained by calling the **GetCurrentConsoleFont** function.

Return value

If the function succeeds, the return value is a **COORD** structure that contains the width and height of each character in the font, in logical units. The **X** member contains the width, while the **Y** member contains the height.

If the function fails, the width and the height are zero. To get extended error information, call **GetLastError**.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Screen Buffers

COORD

GetCurrentConsoleFont

GetConsoleHistoryInfo function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the history settings for the calling process's console.

Syntax

```
BOOL WINAPI GetConsoleHistoryInfo(
  _Out_ PCONSOLE_HISTORY_INFO lpConsoleHistoryInfo
);
```

Parameters

lpConsoleHistoryInfo [out]
A pointer to a CONSOLE_HISTORY_INFO structure that receives the history settings for the calling process's console.

Return value

If the function succeeds the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the calling process is not a console process, the function fails and sets the last error to ERROR_ACCESS_DENIED.

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application acting as a shell or interpreter is expected to maintain its own user-convenience functionality like line reading and manipulation behavior including aliases and command history. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

CONSOLE_HISTORY_INFO

SetConsoleHistoryInfo

GetConsoleMode function

5/18/2021 • 6 minutes to read • Edit Online

Retrieves the current input mode of a console's input buffer or the current output mode of a console screen buffer.

Syntax

```
BOOL WINAPI GetConsoleMode(
  _In_  HANDLE  hConsoleHandle,
  _Out_ LPDWORD lpMode
);
```

Parameters

hConsoleHandle [in]
A handle to the console input buffer or the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpMode [out]
A pointer to a variable that receives the current mode of the specified buffer.

If the *hConsoleHandle* parameter is an input handle, the mode can be one or more of the following values. When a console is created, all input modes except **ENABLE_WINDOW_INPUT** and **ENABLE_VIRTUAL_TERMINAL_INPUT** are enabled by default.

VALUE	MEANING
ENABLE_ECHO_INPUT 0x0004	Characters read by the **ReadFile** or **ReadConsole** function are written to the active screen buffer as they are typed into the console. This mode can be used only if the **ENABLE_LINE_INPUT** mode is also enabled.
ENABLE_INSERT_MODE 0x0020	When enabled, text entered in a console window will be inserted at the current cursor location and all text following that location will not be overwritten. When disabled, all following text will be overwritten.
ENABLE_LINE_INPUT 0x0002	The **ReadFile** or **ReadConsole** function returns only when a carriage return character is read. If this mode is disabled, the functions return when one or more characters are available.
ENABLE_MOUSE_INPUT 0x0010	If the mouse pointer is within the borders of the console window and the window has the keyboard focus, mouse events generated by mouse movement and button presses are placed in the input buffer. These events are discarded by **ReadFile** or **ReadConsole**, even when this mode is enabled.

VALUE	MEANING
ENABLE_PROCESSED_INPUT 0x0001	CTRL+C is processed by the system and is not placed in the input buffer. If the input buffer is being read by **ReadFile** or **ReadConsole**, other control keys are processed by the system and are not returned in the **ReadFile** or **ReadConsole** buffer. If the **ENABLE_LINE_INPUT** mode is also enabled, backspace, carriage return, and line feed characters are handled by the system.
ENABLE_QUICK_EDIT_MODE 0x0040	This flag enables the user to use the mouse to select and edit text. To enable this mode, use `ENABLE_QUICK_EDIT_MODE | ENABLE_EXTENDED_FLAGS`. To disable this mode, use **ENABLE_EXTENDED_FLAGS** without this flag.
ENABLE_WINDOW_INPUT 0x0008	User interactions that change the size of the console screen buffer are reported in the console's input buffer. Information about these events can be read from the input buffer by applications using the **ReadConsoleInput** function, but not by those using **ReadFile** or **ReadConsole**.
ENABLE_VIRTUAL_TERMINAL_INPUT 0x0200	Setting this flag directs the Virtual Terminal processing engine to convert user input received by the console window into **Console Virtual Terminal Sequences** that can be retrieved by a supporting application through **WriteFile** or **WriteConsole** functions. The typical usage of this flag is intended in conjunction with ENABLE_VIRTUAL_TERMINAL_PROCESSING on the output handle to connect to an application that communicates exclusively via virtual terminal sequences.

If the *hConsoleHandle* parameter is a screen buffer handle, the mode can be one or more of the following values. When a screen buffer is created, both output modes are enabled by default.

VALUE	MEANING
ENABLE_PROCESSED_OUTPUT 0x0001	Characters written by the **WriteFile** or **WriteConsole** function or echoed by the **ReadFile** or **ReadConsole** function are parsed for ASCII control sequences, and the correct action is performed. Backspace, tab, bell, carriage return, and line feed characters are processed.
ENABLE_WRAP_AT_EOL_OUTPUT 0x0002	When writing with **WriteFile** or **WriteConsole** or echoing with **ReadFile** or **ReadConsole**, the cursor moves to the beginning of the next row when it reaches the end of the current row. This causes the rows displayed in the console window to scroll up automatically when the cursor advances beyond the last row in the window. It also causes the contents of the console screen buffer to scroll up (../discarding the top row of the console screen buffer) when the cursor advances beyond the last row in the console screen buffer. If this mode is disabled, the last character in the row is overwritten with any subsequent characters.

VALUE	MEANING
ENABLE_VIRTUAL_TERMINAL_PROCESSING 0x0004	When writing with **WriteFile** or **WriteConsole**, characters are parsed for VT100 and similar control character sequences that control cursor movement, color/font mode, and other operations that can also be performed via the existing Console APIs. For more information, see **Console Virtual Terminal Sequences**.
DISABLE_NEWLINE_AUTO_RETURN 0x0008	When writing with **WriteFile** or **WriteConsole**, this adds an additional state to end-of-line wrapping that can delay the cursor move and buffer scroll operations. Normally when **ENABLE_WRAP_AT_EOL_OUTPUT** is set and text reaches the end of the line, the cursor will immediately move to the next line and the contents of the buffer will scroll up by one line. In contrast with this flag set, the scroll operation and cursor move is delayed until the next character arrives. The written character will be printed in the final position on the line and the cursor will remain above this character as if **ENABLE_WRAP_AT_EOL_OUTPUT** was off, but the next printable character will be printed as if **ENABLE_WRAP_AT_EOL_OUTPUT** is on. No overwrite will occur. Specifically, the cursor quickly advances down to the following line, a scroll is performed if necessary, the character is printed, and the cursor advances one more position. The typical usage of this flag is intended in conjunction with setting **ENABLE_VIRTUAL_TERMINAL_PROCESSING** to better emulate a terminal emulator where writing the final character on the screen (../in the bottom right corner) without triggering an immediate scroll is the desired behavior.
ENABLE_LVB_GRID_WORLDWIDE 0x0010	The APIs for writing character attributes including **WriteConsoleOutput** and **WriteConsoleOutputAttribute** allow the usage of flags from **character attributes** to adjust the color of the foreground and background of text. Additionally, a range of DBCS flags was specified with the COMMON_LVB prefix. Historically, these flags only functioned in DBCS code pages for Chinese, Japanese, and Korean languages. With exception of the leading byte and trailing byte flags, the remaining flags describing line drawing and reverse video (../swap foreground and background colors) can be useful for other languages to emphasize portions of output. Setting this console mode flag will allow these attributes to be used in every code page on every language. It is off by default to maintain compatibility with known applications that have historically taken advantage of the console ignoring these flags on non-CJK machines to store bits in these fields for their own purposes or by accident. Note that using the ENABLE_VIRTUAL_TERMINAL_PROCESSING mode can result in LVB grid and reverse video flags being set while this flag is still off if the attached application requests underlining or inverse video via **Console Virtual Terminal Sequences**.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A console consists of an input buffer and one or more screen buffers. The mode of a console buffer determines how the console behaves during input or output (I/O) operations. One set of flag constants is used with input handles, and another set is used with screen buffer (output) handles. Setting the output modes of one screen buffer does not affect the output modes of other screen buffers.

The **ENABLE_LINE_INPUT** and **ENABLE_ECHO_INPUT** modes only affect processes that use **ReadFile** or **ReadConsole** to read from the console's input buffer. Similarly, the **ENABLE_PROCESSED_INPUT** mode primarily affects **ReadFile** and **ReadConsole** users, except that it also determines whether CTRL+C input is reported in the input buffer (to be read by the **ReadConsoleInput** function) or is passed to a function defined by the application.

The **ENABLE_WINDOW_INPUT** and **ENABLE_MOUSE_INPUT** modes determine whether user interactions involving window resizing and mouse actions are reported in the input buffer or discarded. These events can be read by **ReadConsoleInput**, but they are always filtered by **ReadFile** and **ReadConsole**.

The **ENABLE_PROCESSED_OUTPUT** and **ENABLE_WRAP_AT_EOL_OUTPUT** modes only affect processes using **ReadFile** or **ReadConsole** and **WriteFile** or **WriteConsole**.

To change a console's I/O modes, call **SetConsoleMode** function.

Examples

For an example, see Reading Input Buffer Events.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Modes

ReadConsole

ReadConsoleInput

ReadFile

SetConsoleMode

WriteConsole

WriteFile

GetConsoleOriginalTitle function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the original title for the current console window.

Syntax

```
DWORD WINAPI GetConsoleOriginalTitle(
  _Out_ LPTSTR lpConsoleTitle,
  _In_  DWORD  nSize
);
```

Parameters

lpConsoleTitle [out]
A pointer to a buffer that receives a null-terminated string containing the original title.

nSize [in]
The size of the *lpConsoleTitle* buffer, in characters.

Return value

If the function succeeds, the return value is the length of the string copied to the buffer, in characters.

If the buffer is not large enough to store the title, the return value is zero and GetLastError returns ERROR_SUCCESS.

If the function fails, the return value is zero and GetLastError returns the error code.

Remarks

To set the title for a console window, use the SetConsoleTitle function. To retrieve the current title string, use the GetConsoleTitle function.

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or later. For more information, see Using the Windows Headers.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**GetConsoleOriginalTitleW** (Unicode) and **GetConsoleOriginalTitleA** (ANSI)

See also

Console Functions

GetConsoleTitle

SetConsoleTitle

GetConsoleOutputCP function

5/18/2021 • 2 minutes to read • Edit Online

Retrieves the output code page used by the console associated with the calling process. A console uses its output code page to translate the character values written by the various output functions into the images displayed in the console window.

Syntax

```
UINT WINAPI GetConsoleOutputCP(void);
```

Parameters

This function has no parameters.

Return value

The return value is a code that identifies the code page. For a list of identifiers, see Code Page Identifiers.

If the return value is zero, the function has failed. To get extended error information, call **GetLastError**.

Remarks

A code page maps 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages. To retrieve more information about a code page, including it's name, see the **GetCPInfoEx** function.

To set a console's output code page, use the **SetConsoleOutputCP** function. To set and query a console's input code page, use the **SetConsoleCP** and **GetConsoleCP** functions.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Code Pages

Console Functions

GetConsoleCP

SetConsoleCP

SetConsoleOutputCP

GetConsoleProcessList function

5/18/2021 • 2 minutes to read • Edit Online

Retrieves a list of the processes attached to the current console.

Syntax

```
DWORD WINAPI GetConsoleProcessList(
  _Out_ LPDWORD lpdwProcessList,
  _In_  DWORD   dwProcessCount
);
```

Parameters

lpdwProcessList [out]
A pointer to a buffer that receives an array of process identifiers upon success. This must be a valid buffer and cannot be `NULL`. The buffer must have space to receive at least 1 returned process id.

dwProcessCount [in]
The maximum number of process identifiers that can be stored in the *lpdwProcessList* buffer. This must be greater than 0.

Return value

If the function succeeds, the return value is less than or equal to *dwProcessCount* and represents the number of process identifiers stored in the *lpdwProcessList* buffer.

If the buffer is too small to hold all the valid process identifiers, the return value is the required number of array elements. The function will have stored no identifiers in the buffer. In this situation, use the return value to allocate a buffer that is large enough to store the entire list and call the function again.

If the return value is zero, the function has failed, because every console has at least one process associated with it. To get extended error information, call **GetLastError**.

If a `NULL` process list was provided or the process count was 0, the call will return 0 and `GetLastError` will return `ERROR_INVALID_PARAMETER`. Please provide a buffer of at least one element to call this function. Allocate a larger buffer and call again if the return code is larger than the length of the provided buffer.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0501 or later. For more information, see Using the Windows Headers.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems. This state is only relevant to the local user, session, and privilege context. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

AttachConsole

Console Functions

GetConsoleScreenBufferInfo function

5/18/2021 • 2 minutes to read • Edit Online

Retrieves information about the specified console screen buffer.

Syntax

```
BOOL WINAPI GetConsoleScreenBufferInfo(
  _In_  HANDLE                      hConsoleOutput,
  _Out_ PCONSOLE_SCREEN_BUFFER_INFO lpConsoleScreenBufferInfo
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpConsoleScreenBufferInfo [out]
A pointer to a **CONSOLE_SCREEN_BUFFER_INFO** structure that receives the console screen buffer information.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The rectangle returned in the **srWindow** member of the **CONSOLE_SCREEN_BUFFER_INFO** structure can be modified and then passed to the **SetConsoleWindowInfo** function to scroll the console screen buffer in the window, to change the size of the window, or both.

All coordinates returned in the **CONSOLE_SCREEN_BUFFER_INFO** structure are in character-cell coordinates, where the origin (0, 0) is at the upper-left corner of the console screen buffer.

> **TIP**
> This API does not have a **virtual terminal** equivalent. Its use may still be required for applications that are attempting to draw columns, grids, or fill the display to retrieve the window size. This window state is managed by the TTY/PTY/Pseudoconsole outside of the normal stream flow and is generally considered a user privilege not adjustable by the client application. Updates can be received on **ReadConsoleInput**.

Examples

For an example, see Scrolling a Screen Buffer's Window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

CONSOLE_SCREEN_BUFFER_INFO

GetLargestConsoleWindowSize

SetConsoleCursorPosition

SetConsoleScreenBufferSize

SetConsoleWindowInfo

Window and Screen Buffer Size

GetConsoleScreenBufferInfoEx function

5/18/2021 • 2 minutes to read • Edit Online

Retrieves extended information about the specified console screen buffer.

Syntax

```
BOOL WINAPI GetConsoleScreenBufferInfoEx(
  _In_  HANDLE                      hConsoleOutput,
  _Out_ PCONSOLE_SCREEN_BUFFER_INFOEX lpConsoleScreenBufferInfoEx
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpConsoleScreenBufferInfoEx [out]
A **CONSOLE_SCREEN_BUFFER_INFOEX** structure that receives the requested console screen buffer information.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The rectangle returned in the **srWindow** member of the **CONSOLE_SCREEN_BUFFER_INFOEX** structure can be modified and then passed to the **SetConsoleWindowInfo** function to scroll the console screen buffer in the window, to change the size of the window, or both.

All coordinates returned in the **CONSOLE_SCREEN_BUFFER_INFOEX** structure are in character-cell coordinates, where the origin (0, 0) is at the upper-left corner of the console screen buffer.

> **TIP**
> This API does not have a **virtual terminal** equivalent. Its use may still be required for applications that are attempting to draw columns, grids, or fill the display to retrieve the window size. This window state is managed by the TTY/PTY/Pseudoconsole outside of the normal stream flow and is generally considered a user privilege not adjustable by the client application. Updates can be received on **ReadConsoleInput**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

CONSOLE_SCREEN_BUFFER_INFOEX

SetConsoleScreenBufferInfoEx

GetConsoleSelectionInfo function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves information about the current console selection.

Syntax

```
BOOL WINAPI GetConsoleSelectionInfo(
  _Out_ PCONSOLE_SELECTION_INFO lpConsoleSelectionInfo
);
```

Parameters

lpConsoleSelectionInfo [out]
A pointer to a CONSOLE_SELECTION_INFO structure that receives the selection information.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]

Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Selection

CONSOLE_SELECTION_INFO

GetConsoleTitle function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the title for the current console window.

Syntax

```
DWORD WINAPI GetConsoleTitle(
  _Out_ LPTSTR lpConsoleTitle,
  _In_  DWORD  nSize
);
```

Parameters

lpConsoleTitle [out]
A pointer to a buffer that receives a null-terminated string containing the title. If the buffer is too small to store the title, the function stores as many characters of the title as will fit in the buffer, ending with a null terminator.

nSize [in]
The size of the buffer pointed to by the *lpConsoleTitle* parameter, in characters.

Return value

If the function succeeds, the return value is the length of the console window's title, in characters.

If the function fails, the return value is zero and GetLastError returns the error code.

Remarks

To set the title for a console window, use the SetConsoleTitle function. To retrieve the original title string, use the GetConsoleOriginalTitle function.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the SetConsoleCP or SetConsoleOutputCP functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Examples

For an example, see **SetConsoleTitle**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**GetConsoleTitleW** (Unicode) and **GetConsoleTitleA** (ANSI)

See also

Console Functions

GetConsoleOriginalTitle

SetConsoleCP

SetConsoleOutputCP

SetConsoleTitle

GetConsoleWindow function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the window handle used by the console associated with the calling process.

Syntax

```
HWND WINAPI GetConsoleWindow(void);
```

Parameters

This function has no parameters.

Return value

The return value is a handle to the window used by the console associated with the calling process or **NULL** if there is no such associated console.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems. This state is only relevant to the local user, session, and privilege context. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

For an application that is hosted inside a **pseudoconsole** session, this function returns a window handle for message queue purposes only. The associated window is not displayed locally as the *pseudoconsole* is serializing all actions to a stream for presentation on another terminal window elsewhere.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

GetCurrentConsoleFont function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves information about the current console font.

Syntax

```
BOOL WINAPI GetCurrentConsoleFont(
  _In_  HANDLE              hConsoleOutput,
  _In_  BOOL                bMaximumWindow,
  _Out_ PCONSOLE_FONT_INFO lpConsoleCurrentFont
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

bMaximumWindow [in]
If this parameter is **TRUE**, font information is retrieved for the maximum window size. If this parameter is **FALSE**, font information is retrieved for the current window size.

lpConsoleCurrentFont [out]
A pointer to a **CONSOLE_FONT_INFO** structure that receives the requested font information.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Screen Buffers

CONSOLE_FONT_INFO

GetConsoleFontSize

GetCurrentConsoleFontEx function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
>
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves extended information about the current console font.

Syntax

```
BOOL WINAPI GetCurrentConsoleFontEx(
  _In_  HANDLE               hConsoleOutput,
  _In_  BOOL                 bMaximumWindow,
  _Out_ PCONSOLE_FONT_INFOEX lpConsoleCurrentFontEx
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

bMaximumWindow [in]
If this parameter is **TRUE**, font information is retrieved for the maximum window size. If this parameter is **FALSE**, font information is retrieved for the current window size.

lpConsoleCurrentFontEx [out]
A pointer to a **CONSOLE_FONT_INFOEX** structure that receives the requested font information.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

> **TIP**
>
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

CONSOLE_FONT_INFOEX

GetLargestConsoleWindowSize function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the size of the largest possible console window, based on the current font and the size of the display.

Syntax

```
COORD WINAPI GetLargestConsoleWindowSize(
  _In_ HANDLE hConsoleOutput
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer.

Return value

If the function succeeds, the return value is a **COORD** structure that specifies the number of character cell columns (**X** member) and rows (**Y** member) in the largest possible console window. Otherwise, the members of the structure are zero.

To get extended error information, call **GetLastError**.

Remarks

The function does not take into consideration the size of the console screen buffer, which means that the window size returned may be larger than the size of the console screen buffer. The **GetConsoleScreenBufferInfo** function can be used to determine the maximum size of the console window, given the current screen buffer size, the current font, and the display size.

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

COORD

GetConsoleScreenBufferInfo

SetConsoleWindowInfo

Window and Screen Buffer Size

GetNumberOfConsoleInputEvents function

5/18/2021 • 2 minutes to read • Edit Online

Retrieves the number of unread input records in the console's input buffer.

Syntax

```
BOOL WINAPI GetNumberOfConsoleInputEvents(
  _In_  HANDLE  hConsoleInput,
  _Out_ LPDWORD lpcNumberOfEvents
);
```

Parameters

hConsoleInput [in]
A handle to the console input buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpcNumberOfEvents [out]
A pointer to a variable that receives the number of unread input records in the console's input buffer.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNumberOfConsoleInputEvents** function reports the total number of unread input records in the input buffer, including keyboard, mouse, and window-resizing input records. Processes using the **ReadFile** or **ReadConsole** function can only read keyboard input. Processes using the **ReadConsoleInput** function can read all types of input records.

A process can specify a console input buffer handle in one of the wait functions to determine when there is unread console input. When the input buffer is not empty, the state of a console input buffer handle is signaled.

To read input records from a console input buffer without affecting the number of unread records, use the **PeekConsoleInput** function. To discard all unread records in a console's input buffer, use the **FlushConsoleInputBuffer** function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

FlushConsoleInputBuffer

Low-Level Console Input Functions

PeekConsoleInput

ReadConsole

ReadConsoleInput

ReadFile

GetNumberOfConsoleMouseButtons function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
>
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Retrieves the number of buttons on the mouse used by the current console.

Syntax

```
BOOL WINAPI GetNumberOfConsoleMouseButtons(
  _Out_ LPDWORD lpNumberOfMouseButtons
);
```

Parameters

lpNumberOfMouseButtons [out]
A pointer to a variable that receives the number of mouse buttons.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When a console receives mouse input, an **INPUT_RECORD** structure containing a **MOUSE_EVENT_RECORD** structure is placed in the console's input buffer. The **dwButtonState** member of **MOUSE_EVENT_RECORD** has a bit indicating the state of each mouse button. The bit is 1 if the button is down and 0 if the button is up. To determine the number of bits that are significant, use **GetNumberOfConsoleMouseButtons**.

> **TIP**
>
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems. This state is only relevant to the local user, session, and privilege context. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Input Buffer

ReadConsoleInput

INPUT_RECORD

MOUSE_EVENT_RECORD

GetStdHandle function

9/16/2021 • 5 minutes to read • Edit Online

Retrieves a handle to the specified standard device (standard input, standard output, or standard error).

Syntax

```
HANDLE WINAPI GetStdHandle(
  _In_ DWORD nStdHandle
);
```

Parameters

nStdHandle [in]
The standard device. This parameter can be one of the following values.

VALUE	MEANING
STD_INPUT_HANDLE `((DWORD)-10)`	The standard input device. Initially, this is the console input buffer, `CONIN$`.
STD_OUTPUT_HANDLE `((DWORD)-11)`	The standard output device. Initially, this is the active console screen buffer, `CONOUT$`.
STD_ERROR_HANDLE `((DWORD)-12)`	The standard error device. Initially, this is the active console screen buffer, `CONOUT$`.

NOTE

The values for these constants are unsigned numbers, but are defined in the header files as a cast from a signed number and take advantage of the C compiler rolling them over to just under the maximum 32-bit value. When interfacing with these handles in a language that does not parse the headers and is re-defining the constants, please be aware of this constraint. As an example, `((DWORD)-10)` is actually the unsigned number `4294967286`.

Return value

If the function succeeds, the return value is a handle to the specified device, or a redirected handle set by a previous call to **SetStdHandle**. The handle has **GENERIC_READ** and **GENERIC_WRITE** access rights, unless the application has used **SetStdHandle** to set a standard handle with lesser access.

TIP

It is not required to dispose of this handle with **CloseHandle** when done. See **Remarks** for more information.

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call **GetLastError**.

If an application does not have associated standard handles, such as a service running on an interactive desktop,

and has not redirected them, the return value is **NULL**.

Remarks

Handles returned by **GetStdHandle** can be used by applications that need to read from or write to the console. When a console is created, the standard input handle is a handle to the console's input buffer, and the standard output and standard error handles are handles of the console's active screen buffer. These handles can be used by the **ReadFile** and **WriteFile** functions, or by any of the console functions that access the console input buffer or a screen buffer (for example, the **ReadConsoleInput**, **WriteConsole**, or **GetConsoleScreenBufferInfo** functions).

The standard handles of a process may be redirected by a call to **SetStdHandle**, in which case **GetStdHandle** returns the redirected handle. If the standard handles have been redirected, you can specify the `CONIN$` value in a call to the **CreateFile** function to get a handle to a console's input buffer. Similarly, you can specify the `CONOUT$` value to get a handle to a console's active screen buffer.

The standard handles of a process on entry of the main method are dictated by the configuration of the **/SUBSYSTEM** flag passed to the linker when the application was built. Specifying **/SUBSYSTEM:CONSOLE** requests that the operating system fill the handles with a console session on startup, if the parent didn't already fill the standard handle table by inheritance. On the contrary, **/SUBSYSTEM:WINDOWS** implies that the application does not need a console and will likely not be making use of the standard handles. More information on handle inheritance can be found in the documentation for **STARTF_USESTDHANDLES**.

Some applications operate outside the boundaries of their declared subsystem; for instance, a **/SUBSYSTEM:WINDOWS** application might check/use standard handles for logging or debugging purposes but operate normally with a graphical user interface. These applications will need to carefully probe the state of standard handles on startup and make use of **AttachConsole**, **AllocConsole**, and **FreeConsole** to add/remove a console if desired.

Some applications may also vary their behavior on the type of inherited handle. Disambiguating the type between console, pipe, file, and others can be performed with **GetFileType**.

Handle disposal

It is not required to **CloseHandle** when done with the handle retrieved from **GetStdHandle**. The returned value is simply a copy of the value stored in the process table. The process itself is generally considered the owner of these handles and their lifetime. Each handle is placed in the table on creation depending on the inheritance and launch specifics of the **CreateProcess** call and will be freed when the process is destroyed.

Manual manipulation of the lifetime of these handles may be desirable for an application intentionally trying to replace them or block other parts of the process from using them. As a `HANDLE` can be cached by running code, that code will not necessarily pick up changes made via **SetStdHandle**. Closing the handle explicitly via **CloseHandle** will close it process-wide and the next usage of any cached reference will encounter an error.

Guidance for replacing a standard handle in the process table would be to get the existing `HANDLE` from the table with **GetStdHandle**, use **SetStdHandle** to place a new `HANDLE` in that is opened with **CreateFile** (or a similar function), then to close the retrieved handle.

There is no validation of the values stored as handles in the process table by either the **GetStdHandle** or **SetStdHandle** functions. Validation is performed at the time of the actual read/write operation such as **ReadFile** or **WriteFile**.

Attach/detach behavior

When attaching to a new console, standard handles are always replaced with console handles unless **STARTF_USESTDHANDLES** was specified during process creation.

If the existing value of the standard handle is **NULL**, or the existing value of the standard handle looks like a

console pseudohandle, the handle is replaced with a console handle.

When a parent uses both **CREATE_NEW_CONSOLE** and **STARTF_USESTDHANDLES** to create a console process, standard handles will not be replaced unless the existing value of the standard handle is **NULL** or a console pseudohandle.

> **NOTE**
> Console processes *must* start with the standard handles filled or they will be filled automatically with appropriate handles to a new console. Graphical user interface (GUI) applications can be started without the standard handles and they will not be automatically filled.

Examples

For an example, see Reading Input Buffer Events.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ProcessEnv.h (via Winbase.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Handles

CreateFile

GetConsoleScreenBufferInfo

ReadConsoleInput

ReadFile

SetStdHandle

WriteConsole

WriteFile

HandlerRoutine callback function

5/18/2021 • 3 minutes to read • Edit Online

An application-defined function used with the **SetConsoleCtrlHandler** function. A console process uses this function to handle control signals received by the process. When the signal is received, the system creates a new thread in the process to execute the function.

The **PHANDLER_ROUTINE** type defines a pointer to this callback function. **HandlerRoutine** is a placeholder for the application-defined function name.

Syntax

```
BOOL WINAPI HandlerRoutine(
  _In_ DWORD dwCtrlType
);
```

Parameters

dwCtrlType [in]
The type of control signal received by the handler. This parameter can be one of the following values.

VALUE	MEANING
CTRL_C_EVENT 0	A CTRL+C signal was received, either from keyboard input or from a signal generated by the **GenerateConsoleCtrlEvent** function.
CTRL_BREAK_EVENT 1	A CTRL+BREAK signal was received, either from keyboard input or from a signal generated by **GenerateConsoleCtrlEvent**.
CTRL_CLOSE_EVENT 2	A signal that the system sends to all processes attached to a console when the user closes the console (either by clicking **Close** on the console window's window menu, or by clicking the **End Task** button command from Task Manager).
CTRL_LOGOFF_EVENT 5	A signal that the system sends to all console processes when a user is logging off. This signal does not indicate which user is logging off, so no assumptions can be made. Note that this signal is received only by services. Interactive applications are terminated at logoff, so they are not present when the system sends this signal.

VALUE	MEANING
CTRL_SHUTDOWN_EVENT 6	A signal that the system sends when the system is shutting down. Interactive applications are not present by the time the system sends this signal, therefore it can be received only be services in this situation. Services also have their own notification mechanism for shutdown events. For more information, see **Handler**. This signal can also be generated by an application using **GenerateConsoleCtrlEvent**.

Return value

If the function handles the control signal, it should return **TRUE**. If it returns **FALSE**, the next handler function in the list of handlers for this process is used.

Remarks

Because the system creates a new thread in the process to execute the handler function, it is possible that the handler function will be terminated by another thread in the process. Be sure to synchronize threads in the process with the thread for the handler function.

Each console process has its own list of **HandlerRoutine** functions. Initially, this list contains only a default handler function that calls **ExitProcess**. A console process adds or removes additional handler functions by calling the **SetConsoleCtrlHandler** function, which does not affect the list of handler functions for other processes. When a console process receives any of the control signals, its handler functions are called on a last-registered, first-called basis until one of the handlers returns **TRUE**. If none of the handlers returns **TRUE**, the default handler is called.

The **CTRL_CLOSE_EVENT**, **CTRL_LOGOFF_EVENT**, and **CTRL_SHUTDOWN_EVENT** signals give the process an opportunity to clean up before termination. A **HandlerRoutine** can perform any necessary cleanup, then take one of the following actions:

- Call the **ExitProcess** function to terminate the process.
- Return **FALSE**. If none of the registered handler functions returns **TRUE**, the default handler terminates the process.
- Return **TRUE**. In this case, no other handler functions are called and the system terminates the process.

A process can use the **SetProcessShutdownParameters** function to prevent the system from displaying a dialog box to the user during logoff or shutdown. In this case, the system terminates the process when **HandlerRoutine** returns **TRUE** or when the time-out period elapses.

When a console application is run as a service, it receives a modified default console control handler. This modified handler does not call **ExitProcess** when processing the **CTRL_LOGOFF_EVENT** and **CTRL_SHUTDOWN_EVENT** signals. This allows the service to continue running after the user logs off. If the service installs its own console control handler, this handler is called before the default handler. If the installed handler calls **ExitProcess** when processing the **CTRL_LOGOFF_EVENT** signal, the service exits when the user logs off.

Note that a third-party library or DLL can install a console control handler for your application. If it does, this handler overrides the default handler, and can cause the application to exit when the user logs off.

Timeouts

EVENT	CIRCUMSTANCES	TIMEOUT
`CTRL_CLOSE_EVENT`	*any*	system parameter `SPI_GETHUNGAPPTIMEOUT`, 5000ms
`CTRL_LOGOFF_EVENT`	*quick*[1]	registry key `CriticalAppShutdownTimeout` or 500ms
`CTRL_LOGOFF_EVENT`	*none of the above*	system parameter `SPI_GETWAITTOKILLTIMEOUT`, 5000ms
`CTRL_SHUTDOWN_EVENT`	**service process**	system parameter `SPI_GETWAITTOKILLSERVICETIMEOUT`, 20000ms
`CTRL_SHUTDOWN_EVENT`	*quick*[1]	registry key `CriticalAppShutdownTimeout` or 500ms
`CTRL_SHUTDOWN_EVENT`	*none of the above*	system parameter `SPI_GETWAITTOKILLTIMEOUT`, 5000ms
`CTRL_C`, `CTRL_BREAK`	*any*	**no timeout**

[1]: "quick" events are never used, but there's still code to support them.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)

See also

Console Control Handlers

Console Functions

ExitProcess

GenerateConsoleCtrlEvent

GetProcessShutdownParameters

SetConsoleCtrlHandler

SetProcessShutdownParameters

PeekConsoleInput function

8/3/2021 • 2 minutes to read • Edit Online

Reads data from the specified console input buffer without removing it from the buffer.

Syntax

```
BOOL WINAPI PeekConsoleInput(
  _In_  HANDLE        hConsoleInput,
  _Out_ PINPUT_RECORD lpBuffer,
  _In_  DWORD         nLength,
  _Out_ LPDWORD       lpNumberOfEventsRead
);
```

Parameters

hConsoleInput [in]
A handle to the console input buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpBuffer [out]
A pointer to an array of **INPUT_RECORD** structures that receives the input buffer data.

nLength [in]
The size of the array pointed to by the *lpBuffer* parameter, in array elements.

lpNumberOfEventsRead [out]
A pointer to a variable that receives the number of input records read.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the number of records requested exceeds the number of records available in the buffer, the number available is read. If no data is available, the function returns immediately.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**PeekConsoleInputW** (Unicode) and **PeekConsoleInputA** (ANSI)

See also

Console Functions

ReadConsoleInput

SetConsoleCP

SetConsoleOutputCP

WriteConsoleInput

INPUT_RECORD

ReadConsole function

5/18/2021 • 2 minutes to read • Edit Online

Reads character input from the console input buffer and removes it from the buffer.

Syntax

```
BOOL WINAPI ReadConsole(
  _In_     HANDLE  hConsoleInput,
  _Out_    LPVOID  lpBuffer,
  _In_     DWORD   nNumberOfCharsToRead,
  _Out_    LPDWORD lpNumberOfCharsRead,
  _In_opt_ LPVOID  pInputControl
);
```

Parameters

hConsoleInput [in]
A handle to the console input buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpBuffer [out]
A pointer to a buffer that receives the data read from the console input buffer.

nNumberOfCharsToRead [in]
The number of characters to be read. The size of the buffer pointed to by the *lpBuffer* parameter should be at least `nNumberOfCharsToRead * sizeof(TCHAR)` bytes.

lpNumberOfCharsRead [out]
A pointer to a variable that receives the number of characters actually read.

pInputControl [in, optional]
A pointer to a **CONSOLE_READCONSOLE_CONTROL** structure that specifies a control character to signal the end of the read operation. This parameter can be **NULL**.

This parameter requires Unicode input by default. For ANSI mode, set this parameter to **NULL**.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

ReadConsole reads keyboard input from a console's input buffer. It behaves like the **ReadFile** function, except that it can read in either Unicode (wide-character) or ANSI mode. To have applications that maintain a single set of sources compatible with both modes, use **ReadConsole** rather than **ReadFile**. Although **ReadConsole** can only be used with a console input buffer handle, **ReadFile** can be used with other handles (such as files or pipes). **ReadConsole** fails if used with a standard handle that has been redirected to be something other than a console handle.

All of the input modes that affect the behavior of **ReadFile** have the same effect on **ReadConsole**. To retrieve and set the input modes of a console input buffer, use the **GetConsoleMode** and **SetConsoleMode** functions.

If the input buffer contains input events other than keyboard events (such as mouse events or window-resizing events), they are discarded. Those events can only be read by using the **ReadConsoleInput** function.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

The *pInputControl* parameter can be used to enable intermediate wakeups from the read in response to a file-completion control character specified in a **CONSOLE_READCONSOLE_CONTROL** structure. This feature requires command extensions to be enabled, the standard output handle to be a console output handle, and input to be Unicode.

Windows Server 2003 and Windows XP/2000: The intermediate read feature is not supported.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**ReadConsoleW** (Unicode) and **ReadConsoleA** (ANSI)

See also

Console Functions

CONSOLE_READCONSOLE_CONTROL

GetConsoleMode

Input and Output Methods

ReadConsoleInput

ReadFile

SetConsoleCP

SetConsoleMode

SetConsoleOutputCP

WriteConsole

ReadConsoleInput function

5/18/2021 • 2 minutes to read • Edit Online

Reads data from a console input buffer and removes it from the buffer.

Syntax

```
BOOL WINAPI ReadConsoleInput(
  _In_  HANDLE        hConsoleInput,
  _Out_ PINPUT_RECORD lpBuffer,
  _In_  DWORD         nLength,
  _Out_ LPDWORD       lpNumberOfEventsRead
);
```

Parameters

hConsoleInput [in]
A handle to the console input buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpBuffer [out]
A pointer to an array of **INPUT_RECORD** structures that receives the input buffer data.

nLength [in]
The size of the array pointed to by the *lpBuffer* parameter, in array elements.

lpNumberOfEventsRead [out]
A pointer to a variable that receives the number of input records read.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the number of records requested in the *nLength* parameter exceeds the number of records available in the buffer, the number available is read. The function does not return until at least one input record has been read.

A process can specify a console input buffer handle in one of the wait functions to determine when there is unread console input. When the input buffer is not empty, the state of a console input buffer handle is signaled.

To determine the number of unread input records in a console's input buffer, use the **GetNumberOfConsoleInputEvents** function. To read input records from a console input buffer without affecting the number of unread records, use the **PeekConsoleInput** function. To discard all unread records in a console's input buffer, use the **FlushConsoleInputBuffer** function.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

Examples

For an example, see Reading Input Buffer Events.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**ReadConsoleInputW** (Unicode) and **ReadConsoleInputA** (ANSI)

See also

Console Functions

FlushConsoleInputBuffer

GetNumberOfConsoleInputEvents

INPUT_RECORD

Low-Level Console Input Functions

PeekConsoleInput

ReadConsole

ReadFile

SetConsoleCP

SetConsoleOutputCP

WriteConsoleInput

ReadConsoleOutput function

5/18/2021 • 4 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Reads character and color attribute data from a rectangular block of character cells in a console screen buffer, and the function writes the data to a rectangular block at a specified location in the destination buffer.

Syntax

```
BOOL WINAPI ReadConsoleOutput(
  _In_    HANDLE      hConsoleOutput,
  _Out_   PCHAR_INFO  lpBuffer,
  _In_    COORD       dwBufferSize,
  _In_    COORD       dwBufferCoord,
  _Inout_ PSMALL_RECT lpReadRegion
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpBuffer [out]
A pointer to a destination buffer that receives the data read from the console screen buffer. This pointer is treated as the origin of a two-dimensional array of **CHAR_INFO** structures whose size is specified by the *dwBufferSize* parameter.

dwBufferSize [in]
The size of the *lpBuffer* parameter, in character cells. The **X** member of the **COORD** structure is the number of columns; the **Y** member is the number of rows.

dwBufferCoord [in]
The coordinates of the upper-left cell in the *lpBuffer* parameter that receives the data read from the console screen buffer. The **X** member of the **COORD** structure is the column, and the **Y** member is the row.

lpReadRegion [in, out]
A pointer to a **SMALL_RECT** structure. On input, the structure members specify the upper-left and lower-right coordinates of the console screen buffer rectangle from which the function is to read. On output, the structure members specify the actual rectangle that was used.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

ReadConsoleOutput treats the console screen buffer and the destination buffer as two-dimensional arrays (columns and rows of character cells). The rectangle pointed to by the *lpReadRegion* parameter specifies the size and location of the block to be read from the console screen buffer. A destination rectangle of the same size is located with its upper-left cell at the coordinates of the *dwBufferCoord* parameter in the *lpBuffer* array. Data read from the cells in the console screen buffer source rectangle is copied to the corresponding cells in the destination buffer. If the corresponding cell is outside the boundaries of the destination buffer rectangle (whose dimensions are specified by the *dwBufferSize* parameter), the data is not copied.

Cells in the destination buffer corresponding to coordinates that are not within the boundaries of the console screen buffer are left unchanged. In other words, these are the cells for which no screen buffer data is available to be read.

Before **ReadConsoleOutput** returns, it sets the members of the structure pointed to by the *lpReadRegion* parameter to the actual screen buffer rectangle whose cells were copied into the destination buffer. This rectangle reflects the cells in the source rectangle for which there existed a corresponding cell in the destination buffer, because **ReadConsoleOutput** clips the dimensions of the source rectangle to fit the boundaries of the console screen buffer.

If the rectangle specified by *lpReadRegion* lies completely outside the boundaries of the console screen buffer, or if the corresponding rectangle is positioned completely outside the boundaries of the destination buffer, no data is copied. In this case, the function returns with the members of the structure pointed to by the *lpReadRegion* parameter set such that the **Right** member is less than the **Left**, or the **Bottom** member is less than the **Top**. To determine the size of the console screen buffer, use the **GetConsoleScreenBufferInfo** function.

The **ReadConsoleOutput** function has no effect on the console screen buffer's cursor position. The contents of the console screen buffer are not changed by the function.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application is expected to remember its own drawn state for further manipulation. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Examples

For an example, see Reading and Writing Blocks of Characters and Attributes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**ReadConsoleOutputW** (Unicode) and **ReadConsoleOutputA** (ANSI)

See also

Console Functions

Low-Level Console Output Functions

ReadConsoleOutputAttribute

ReadConsoleOutputCharacter

SetConsoleCP

SetConsoleOutputCP

SMALL_RECT

WriteConsoleOutput

CHAR_INFO

COORD

ReadConsoleOutputAttribute function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Copies a specified number of character attributes from consecutive cells of a console screen buffer, beginning at a specified location.

Syntax

```
BOOL WINAPI ReadConsoleOutputAttribute(
  _In_  HANDLE  hConsoleOutput,
  _Out_ LPWORD  lpAttribute,
  _In_  DWORD   nLength,
  _In_  COORD   dwReadCoord,
  _Out_ LPDWORD lpNumberOfAttrsRead
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpAttribute [out]
A pointer to a buffer that receives the attributes being used by the console screen buffer.

For more information, see Character Attributes.

nLength [in]
The number of screen buffer character cells from which to read. The size of the buffer pointed to by the *lpAttribute* parameter should be `nLength * sizeof(WORD)`.

dwReadCoord [in]
The coordinates of the first cell in the console screen buffer from which to read, in characters. The **X** member of the **COORD** structure is the column, and the **Y** member is the row.

lpNumberOfAttrsRead [out]
A pointer to a variable that receives the number of attributes actually read.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the number of attributes to be read from extends beyond the end of the specified screen buffer row, attributes are read from the next row. If the number of attributes to be read from extends beyond the end of the console screen buffer, attributes up to the end of the console screen buffer are read.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application is expected to remember its own drawn state for further manipulation. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

COORD

Low-Level Console Output Functions

ReadConsoleOutput

ReadConsoleOutputCharacter

WriteConsoleOutput

WriteConsoleOutputAttribute

WriteConsoleOutputCharacter

ReadConsoleOutputCharacter function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Copies a number of characters from consecutive cells of a console screen buffer, beginning at a specified location.

Syntax

```
BOOL WINAPI ReadConsoleOutputCharacter(
  _In_  HANDLE  hConsoleOutput,
  _Out_ LPTSTR  lpCharacter,
  _In_  DWORD   nLength,
  _In_  COORD   dwReadCoord,
  _Out_ LPDWORD lpNumberOfCharsRead
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpCharacter [out]
A pointer to a buffer that receives the characters read from the console screen buffer.

nLength [in]
The number of screen buffer character cells from which to read. The size of the buffer pointed to by the *lpCharacter* parameter should be `nLength * sizeof(TCHAR)`.

dwReadCoord [in]
The coordinates of the first cell in the console screen buffer from which to read, in characters. The **X** member of the **COORD** structure is the column, and the **Y** member is the row.

lpNumberOfCharsRead [out]
A pointer to a variable that receives the number of characters actually read.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the number of characters to be read from extends beyond the end of the specified screen buffer row, characters are read from the next row. If the number of characters to be read from extends beyond the end of the console screen buffer, characters up to the end of the console screen buffer are read.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the SetConsoleCP or SetConsoleOutputCP functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application is expected to remember its own drawn state for further manipulation. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**ReadConsoleOutputCharacterW** (Unicode) and **ReadConsoleOutputCharacterA** (ANSI)

See also

Console Functions

COORD

Low-Level Console Output Functions

ReadConsoleOutput

ReadConsoleOutputAttribute

SetConsoleCP

SetConsoleOutputCP

WriteConsoleOutput

WriteConsoleOutputAttribute

WriteConsoleOutputCharacter

ResizePseudoConsole function

5/18/2021 • 2 minutes to read • Edit Online

Resizes the internal buffers for a pseudoconsole to the given size.

Syntax

```
HRESULT WINAPI ResizePseudoConsole(
    _In_ HPCON hPC ,
    _In_ COORD size
);
```

Parameters

hPC [in]
A handle to an active pseudoconsole as opened by CreatePseudoConsole.

size [in]
The dimensions of the window/buffer in count of characters that will be used for the internal buffer of this pseudoconsole.

Return value

Type: **HRESULT**

If this method succeeds, it returns **S_OK**. Otherwise, it returns an **HRESULT** error code.

Remarks

This function can resize the internal buffers in the pseudoconsole session to match the window/buffer size being used for display on the terminal end. This ensures that attached Command-Line Interface (CUI) applications using the Console Functions to communicate will have the correct dimensions returned in their calls.

Requirements

Minimum supported client	Windows 10 October 2018 Update (version 1809) [desktop apps only]
Minimum supported server	Windows Server 2019 [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Pseudoconsoles

CreatePseudoConsole

ClosePseudoConsole

ScrollConsoleScreenBuffer function

5/18/2021 • 3 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Moves a block of data in a screen buffer. The effects of the move can be limited by specifying a clipping rectangle, so the contents of the console screen buffer outside the clipping rectangle are unchanged.

Syntax

```
BOOL WINAPI ScrollConsoleScreenBuffer(
  _In_           HANDLE     hConsoleOutput,
  _In_     const SMALL_RECT *lpScrollRectangle,
  _In_opt_ const SMALL_RECT *lpClipRectangle,
  _In_           COORD      dwDestinationOrigin,
  _In_     const CHAR_INFO  *lpFill
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpScrollRectangle [in]
A pointer to a **SMALL_RECT** structure whose members specify the upper-left and lower-right coordinates of the console screen buffer rectangle to be moved.

lpClipRectangle [in, optional]
A pointer to a **SMALL_RECT** structure whose members specify the upper-left and lower-right coordinates of the console screen buffer rectangle that is affected by the scrolling. This pointer can be **NULL**.

dwDestinationOrigin [in]
A **COORD** structure that specifies the upper-left corner of the new location of the *lpScrollRectangle* contents, in characters.

lpFill [in]
A pointer to a **CHAR_INFO** structure that specifies the character and color attributes to be used in filling the cells within the intersection of *lpScrollRectangle* and *lpClipRectangle* that were left empty as a result of the move.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

ScrollConsoleScreenBuffer copies the contents of a rectangular region of a screen buffer, specified by the *lpScrollRectangle* parameter, to another area of the console screen buffer. The target rectangle has the same dimensions as the *lpScrollRectangle* rectangle with its upper-left corner at the coordinates specified by the *dwDestinationOrigin* parameter. Those parts of *lpScrollRectangle* that do not overlap with the target rectangle are filled in with the character and color attributes specified by the *lpFill* parameter.

The clipping rectangle applies to changes made in both the *lpScrollRectangle* rectangle and the target rectangle. For example, if the clipping rectangle does not include a region that would have been filled by the contents of *lpFill*, the original contents of the region are left unchanged.

If the scroll or target regions extend beyond the dimensions of the console screen buffer, they are clipped. For example, if *lpScrollRectangle* is the region contained by (0,0) and (19,19) and *dwDestinationOrigin* is (10,15), the target rectangle is the region contained by (10,15) and (29,34). However, if the console screen buffer is 50 characters wide and 30 characters high, the target rectangle is clipped to (10,15) and (29,29). Changes to the console screen buffer are also clipped according to *lpClipRectangle*, if the parameter specifies a **SMALL_RECT** structure. If the clipping rectangle is specified as (0,0) and (49,19), only the changes that occur in that region of the console screen buffer are made.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. Use can be approximated with **scroll margins** to fix an area of the screen, **cursor positioning** to set the active position outside the region, and newlines to force text to move. The remaining space can be filled by moving the cursor, **setting graphical attributes**, and writing normal text.

Examples

For an example, see Scrolling a Screen Buffer's Contents.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Unicode and ANSI names	**ScrollConsoleScreenBufferW** (Unicode) and **ScrollConsoleScreenBufferA** (ANSI)

See also

CHAR_INFO

Console Functions

COORD

Scrolling the Screen Buffer

SetConsoleCP

SetConsoleOutputCP

SetConsoleWindowInfo

SMALL_RECT

SetConsoleActiveScreenBuffer function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets the specified screen buffer to be the currently displayed console screen buffer.

Syntax

```
BOOL WINAPI SetConsoleActiveScreenBuffer(
  _In_ HANDLE hConsoleOutput
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A console can have multiple screen buffers. **SetConsoleActiveScreenBuffer** determines which one is displayed. You can write to an inactive screen buffer and then use **SetConsoleActiveScreenBuffer** to display the buffer's contents.

> **TIP**
> This API is not recommended but it does have an approximate **virtual terminal** equivalent in the **alternate screen buffer** sequence. Setting the *alternate screen buffer* can provide an application with a separate, isolated space for drawing over the course of its session runtime while preserving the content that was displayed by the application's invoker. This maintains that drawing information for simple restoration on process exit.

Examples

For an example, see Reading and Writing Blocks of Characters and Attributes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Screen Buffers

CreateConsoleScreenBuffer

SetConsoleCP function

5/18/2021 • 2 minutes to read • Edit Online

Sets the input code page used by the console associated with the calling process. A console uses its input code page to translate keyboard input into the corresponding character value.

Syntax

```
BOOL WINAPI SetConsoleCP(
  _In_ UINT wCodePageID
);
```

Parameters

wCodePageID [in]
The identifier of the code page to be set. For more information, see Remarks.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A code page maps 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages.

To find the code pages that are installed or supported by the operating system, use the **EnumSystemCodePages** function. The identifiers of the code pages available on the local computer are also stored in the registry under the following key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage
```

However, it is better to use **EnumSystemCodePages** to enumerate code pages because the registry can differ in different versions of Windows.

To determine whether a particular code page is valid, use the **IsValidCodePage** function. To retrieve more information about a code page, including its name, use the **GetCPInfoEx** function. For a list of available code page identifiers, see Code Page Identifiers.

To determine a console's current input code page, use the **GetConsoleCP** function. To set and retrieve a console's output code page, use the **SetConsoleOutputCP** and **GetConsoleOutputCP** functions.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Code Pages

Console Functions

GetConsoleCP

GetConsoleOutputCP

SetConsoleOutputCP

SetConsoleCtrlHandler function

9/16/2021 • 4 minutes to read • Edit Online

Adds or removes an application-defined **HandlerRoutine** function from the list of handler functions for the calling process.

If no handler function is specified, the function sets an inheritable attribute that determines whether the calling process ignores CTRL+C signals.

Syntax

```
BOOL WINAPI SetConsoleCtrlHandler(
  _In_opt_ PHANDLER_ROUTINE HandlerRoutine,
  _In_     BOOL             Add
);
```

Parameters

HandlerRoutine [in, optional]
A pointer to the application-defined **HandlerRoutine** function to be added or removed. This parameter can be **NULL**.

Add [in]
If this parameter is **TRUE**, the handler is added; if it is **FALSE**, the handler is removed.

If the *HandlerRoutine* parameter is **NULL**, a **TRUE** value causes the calling process to ignore CTRL+C input, and a **FALSE** value restores normal processing of CTRL+C input. This attribute of ignoring or processing CTRL+C is inherited by child processes.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This function provides a similar notification for console application and services that **WM_QUERYENDSESSION** provides for graphical applications with a message pump. You could also use this function from a graphical application, but there is no guarantee it would arrive before the notification from **WM_QUERYENDSESSION**.

Each console process has its own list of application-defined **HandlerRoutine** functions that handle CTRL+C and CTRL+BREAK signals. The handler functions also handle signals generated by the system when the user closes the console, logs off, or shuts down the system. Initially, the handler list for each process contains only a default handler function that calls the **ExitProcess** function. A console process adds or removes additional handler functions by calling the **SetConsoleCtrlHandler** function, which does not affect the list of handler functions for other processes. When a console process receives any of the control signals, its handler functions are called on a last-registered, first-called basis until one of the handlers returns `TRUE`. If none of the handlers returns `TRUE`, the default handler is called.

Calling **AttachConsole**, **AllocConsole**, or **FreeConsole** will reset the table of control handlers in the client process to its initial state. Handlers must be registered again when the attached console session changes.

For console processes, the CTRL+C and CTRL+BREAK key combinations are typically treated as signals (**CTRL_C_EVENT** and **CTRL_BREAK_EVENT**). When a console window with the keyboard focus receives CTRL+C or CTRL+BREAK, the signal is typically passed to all processes sharing that console.

CTRL+BREAK is always treated as a signal, but typical CTRL+C behavior can be changed in three ways that prevent the handler functions from being called:

- The **SetConsoleMode** function can disable the **ENABLE_PROCESSED_INPUT** mode for a console's input buffer, so CTRL+C is reported as keyboard input rather than as a signal.
- Calling **SetConsoleCtrlHandler** with the **NULL** and **TRUE** arguments causes the calling process to ignore CTRL+C signals. This attribute is inherited by child processes, but it can be enabled or disabled by any process without affecting existing processes.
- If a console process is being debugged and CTRL+C signals have not been disabled, the system generates a **DBG_CONTROL_C** exception. This exception is raised only for the benefit of the debugger, and an application should never use an exception handler to deal with it. If the debugger handles the exception, an application will not notice the CTRL+C, with one exception: alertable waits will terminate. If the debugger passes the exception on unhandled, CTRL+C is passed to the console process and treated as a signal, as previously discussed.

A console process can use the **GenerateConsoleCtrlEvent** function to send a CTRL+C or CTRL+BREAK signal to a console process group.

The system generates **CTRL_CLOSE_EVENT**, **CTRL_LOGOFF_EVENT**, and **CTRL_SHUTDOWN_EVENT** signals when the user closes the console, logs off, or shuts down the system so that the process has an opportunity to clean up before termination. Console functions, or any C run-time functions that call console functions, may not work reliably during processing of any of the three signals mentioned previously. The reason is that some or all of the internal console cleanup routines may have been called before executing the process signal handler.

Windows 7, Windows 8, Windows 8.1 and Windows 10:

If a console application loads the gdi32.dll or user32.dll library, the **HandlerRoutine** function that you specify when you call **SetConsoleCtrlHandler** does not get called for the **CTRL_LOGOFF_EVENT** and **CTRL_SHUTDOWN_EVENT** events. The operating system recognizes processes that load gdi32.dll or user32.dll as Windows applications rather than console applications. This behavior also occurs for console applications that do not call functions in gdi32.dll or user32.dll directly, but do call functions such as Shell functions that do in turn call functions in gdi32.dll or user32.dll.

To receive events when a user signs out or the device shuts down in these circumstances, create a hidden window in your console application, and then handle the **WM_QUERYENDSESSION** and **WM_ENDSESSION** window messages that the hidden window receives. You can create a hidden window by calling the **CreateWindowEx** method with the *dwExStyle* parameter set to 0. An example of this is included with the basic handler example linked below.

Examples

For an example, see Registering a Control Handler Function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	

See also

Console Control Handlers

Console Functions

ExitProcess

GenerateConsoleCtrlEvent

GetConsoleMode

HandlerRoutine

SetConsoleMode

SetConsoleCursorInfo function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets the size and visibility of the cursor for the specified console screen buffer.

Syntax

```
BOOL WINAPI SetConsoleCursorInfo(
  _In_       HANDLE              hConsoleOutput,
  _In_ const CONSOLE_CURSOR_INFO *lpConsoleCursorInfo
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

lpConsoleCursorInfo [in]
A pointer to a **CONSOLE_CURSOR_INFO** structure that provides the new specifications for the console screen buffer's cursor.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When a screen buffer's cursor is visible, its appearance can vary, ranging from completely filling a character cell to showing up as a horizontal line at the bottom of the cell. The **dwSize** member of the **CONSOLE_CURSOR_INFO** structure specifies the percentage of a character cell that is filled by the cursor. If this member is less than 1 or greater than 100, **SetConsoleCursorInfo** fails.

TIP

This API has a **virtual terminal** equivalent in the **cursor visibility** section with the `^[[?25h` and `^[[?25l` sequences.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Screen Buffers

CONSOLE_CURSOR_INFO

GetConsoleCursorInfo

SetConsoleCursorPosition

SetConsoleCursorPosition function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets the cursor position in the specified console screen buffer.

Syntax

```
BOOL WINAPI SetConsoleCursorPosition(
  _In_ HANDLE hConsoleOutput,
  _In_ COORD  dwCursorPosition
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

dwCursorPosition [in]
A **COORD** structure that specifies the new cursor position, in characters. The coordinates are the column and row of a screen buffer character cell. The coordinates must be within the boundaries of the console screen buffer.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The cursor position determines where characters written by the **WriteFile** or **WriteConsole** function, or echoed by the **ReadFile** or **ReadConsole** function, are displayed. To determine the current position of the cursor, use the **GetConsoleScreenBufferInfo** function.

If the new cursor position is not within the boundaries of the console screen buffer's window, the window origin changes to make the cursor visible.

TIP

This API has a **virtual terminal** equivalent in the **simple cursor positioning** and **cursor positioning** sections. Use of the newline, carriage return, backspace, and tab control sequences can also assist with cursor positioning.

Examples

For an example, see Using the High-Level Input and Output Functions.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Screen Buffers

GetConsoleCursorInfo

GetConsoleScreenBufferInfo

ReadConsole

ReadFile

SetConsoleCursorInfo

WriteConsole

WriteFile

SetConsoleDisplayMode function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets the display mode of the specified console screen buffer.

Syntax

```
BOOL WINAPI SetConsoleDisplayMode(
  _In_      HANDLE hConsoleOutput,
  _In_      DWORD  dwFlags,
  _Out_opt_ PCOORD lpNewScreenBufferDimensions
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer.

dwFlags [in]
The display mode of the console. This parameter can be one or more of the following values.

VALUE	MEANING
CONSOLE_FULLSCREEN_MODE 1	Text is displayed in full-screen mode.
CONSOLE_WINDOWED_MODE 2	Text is displayed in a console window.

lpNewScreenBufferDimensions [out, optional]
A pointer to a COORD structure that receives the new dimensions of the screen buffer, in characters.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Modes

GetConsoleDisplayMode

SetConsoleHistoryInfo function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets the history settings for the calling process's console.

Syntax

```
BOOL WINAPI SetConsoleHistoryInfo(
  _In_ PCONSOLE_HISTORY_INFO lpConsoleHistoryInfo
);
```

Parameters

lpConsoleHistoryInfo [in]
A pointer to a CONSOLE_HISTORY_INFO structure that contains the history settings for the process's console.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the calling process is not a console process, the function fails and sets the last error code to ERROR_ACCESS_DENIED.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the individual client application acting as a shell or interpreter is expected to maintain its own user-convenience functionality like line reading and manipulation behavior including aliases and command history. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

CONSOLE_HISTORY_INFO

GetConsoleHistoryInfo

SetConsoleMode function

5/18/2021 • 6 minutes to read • Edit Online

Sets the input mode of a console's input buffer or the output mode of a console screen buffer.

Syntax

```
BOOL WINAPI SetConsoleMode(
  _In_ HANDLE hConsoleHandle,
  _In_ DWORD  dwMode
);
```

Parameters

hConsoleHandle [in]
A handle to the console input buffer or a console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

dwMode [in]
The input or output mode to be set.

If the *hConsoleHandle* parameter is an input handle, the mode can be one or more of the following values. When a console is created, all input modes except **ENABLE_WINDOW_INPUT** and **ENABLE_VIRTUAL_TERMINAL_INPUT** are enabled by default.

VALUE	MEANING
ENABLE_ECHO_INPUT 0x0004	Characters read by the **ReadFile** or **ReadConsole** function are written to the active screen buffer as they are typed into the console. This mode can be used only if the **ENABLE_LINE_INPUT** mode is also enabled.
ENABLE_INSERT_MODE 0x0020	When enabled, text entered in a console window will be inserted at the current cursor location and all text following that location will not be overwritten. When disabled, all following text will be overwritten.
ENABLE_LINE_INPUT 0x0002	The **ReadFile** or **ReadConsole** function returns only when a carriage return character is read. If this mode is disabled, the functions return when one or more characters are available.
ENABLE_MOUSE_INPUT 0x0010	If the mouse pointer is within the borders of the console window and the window has the keyboard focus, mouse events generated by mouse movement and button presses are placed in the input buffer. These events are discarded by **ReadFile** or **ReadConsole**, even when this mode is enabled.

VALUE	MEANING
ENABLE_PROCESSED_INPUT 0x0001	CTRL+C is processed by the system and is not placed in the input buffer. If the input buffer is being read by **ReadFile** or **ReadConsole**, other control keys are processed by the system and are not returned in the **ReadFile** or **ReadConsole** buffer. If the **ENABLE_LINE_INPUT** mode is also enabled, backspace, carriage return, and line feed characters are handled by the system.
ENABLE_QUICK_EDIT_MODE 0x0040	This flag enables the user to use the mouse to select and edit text. To enable this mode, use `ENABLE_QUICK_EDIT_MODE | ENABLE_EXTENDED_FLAGS`. To disable this mode, use **ENABLE_EXTENDED_FLAGS** without this flag.
ENABLE_WINDOW_INPUT 0x0008	User interactions that change the size of the console screen buffer are reported in the console's input buffer. Information about these events can be read from the input buffer by applications using the **ReadConsoleInput** function, but not by those using **ReadFile** or **ReadConsole**.
ENABLE_VIRTUAL_TERMINAL_INPUT 0x0200	Setting this flag directs the Virtual Terminal processing engine to convert user input received by the console window into **Console Virtual Terminal Sequences** that can be retrieved by a supporting application through **WriteFile** or **WriteConsole** functions. The typical usage of this flag is intended in conjunction with ENABLE_VIRTUAL_TERMINAL_PROCESSING on the output handle to connect to an application that communicates exclusively via virtual terminal sequences.

If the *hConsoleHandle* parameter is a screen buffer handle, the mode can be one or more of the following values. When a screen buffer is created, both output modes are enabled by default.

VALUE	MEANING
ENABLE_PROCESSED_OUTPUT 0x0001	Characters written by the **WriteFile** or **WriteConsole** function or echoed by the **ReadFile** or **ReadConsole** function are parsed for ASCII control sequences, and the correct action is performed. Backspace, tab, bell, carriage return, and line feed characters are processed.
ENABLE_WRAP_AT_EOL_OUTPUT 0x0002	When writing with **WriteFile** or **WriteConsole** or echoing with **ReadFile** or **ReadConsole**, the cursor moves to the beginning of the next row when it reaches the end of the current row. This causes the rows displayed in the console window to scroll up automatically when the cursor advances beyond the last row in the window. It also causes the contents of the console screen buffer to scroll up (../discarding the top row of the console screen buffer) when the cursor advances beyond the last row in the console screen buffer. If this mode is disabled, the last character in the row is overwritten with any subsequent characters.

VALUE	MEANING
ENABLE_VIRTUAL_TERMINAL_PROCESSING 0x0004	When writing with **WriteFile** or **WriteConsole**, characters are parsed for VT100 and similar control character sequences that control cursor movement, color/font mode, and other operations that can also be performed via the existing Console APIs. For more information, see **Console Virtual Terminal Sequences**.
DISABLE_NEWLINE_AUTO_RETURN 0x0008	When writing with **WriteFile** or **WriteConsole**, this adds an additional state to end-of-line wrapping that can delay the cursor move and buffer scroll operations. Normally when **ENABLE_WRAP_AT_EOL_OUTPUT** is set and text reaches the end of the line, the cursor will immediately move to the next line and the contents of the buffer will scroll up by one line. In contrast with this flag set, the scroll operation and cursor move is delayed until the next character arrives. The written character will be printed in the final position on the line and the cursor will remain above this character as if **ENABLE_WRAP_AT_EOL_OUTPUT** was off, but the next printable character will be printed as if **ENABLE_WRAP_AT_EOL_OUTPUT** is on. No overwrite will occur. Specifically, the cursor quickly advances down to the following line, a scroll is performed if necessary, the character is printed, and the cursor advances one more position. The typical usage of this flag is intended in conjunction with setting **ENABLE_VIRTUAL_TERMINAL_PROCESSING** to better emulate a terminal emulator where writing the final character on the screen (../in the bottom right corner) without triggering an immediate scroll is the desired behavior.
ENABLE_LVB_GRID_WORLDWIDE 0x0010	The APIs for writing character attributes including **WriteConsoleOutput** and **WriteConsoleOutputAttribute** allow the usage of flags from **character attributes** to adjust the color of the foreground and background of text. Additionally, a range of DBCS flags was specified with the COMMON_LVB prefix. Historically, these flags only functioned in DBCS code pages for Chinese, Japanese, and Korean languages. With exception of the leading byte and trailing byte flags, the remaining flags describing line drawing and reverse video (../swap foreground and background colors) can be useful for other languages to emphasize portions of output. Setting this console mode flag will allow these attributes to be used in every code page on every language. It is off by default to maintain compatibility with known applications that have historically taken advantage of the console ignoring these flags on non-CJK machines to store bits in these fields for their own purposes or by accident. Note that using the ENABLE_VIRTUAL_TERMINAL_PROCESSING mode can result in LVB grid and reverse video flags being set while this flag is still off if the attached application requests underlining or inverse video via **Console Virtual Terminal Sequences**.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A console consists of an input buffer and one or more screen buffers. The mode of a console buffer determines how the console behaves during input or output (I/O) operations. One set of flag constants is used with input handles, and another set is used with screen buffer (output) handles. Setting the output modes of one screen buffer does not affect the output modes of other screen buffers.

The **ENABLE_LINE_INPUT** and **ENABLE_ECHO_INPUT** modes only affect processes that use **ReadFile** or **ReadConsole** to read from the console's input buffer. Similarly, the **ENABLE_PROCESSED_INPUT** mode primarily affects **ReadFile** and **ReadConsole** users, except that it also determines whether CTRL+C input is reported in the input buffer (to be read by the **ReadConsoleInput** function) or is passed to a function defined by the application.

The **ENABLE_WINDOW_INPUT** and **ENABLE_MOUSE_INPUT** modes determine whether user interactions involving window resizing and mouse actions are reported in the input buffer or discarded. These events can be read by **ReadConsoleInput**, but they are always filtered by **ReadFile** and **ReadConsole**.

The **ENABLE_PROCESSED_OUTPUT** and **ENABLE_WRAP_AT_EOL_OUTPUT** modes only affect processes using **ReadFile** or **ReadConsole** and **WriteFile** or **WriteConsole**.

To determine the current mode of a console input buffer or a screen buffer, use the **GetConsoleMode** function.

Examples

For an example, see Reading Input Buffer Events.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Modes

GetConsoleMode

HandlerRoutine

ReadConsole

ReadConsoleInput

ReadFile

WriteConsole

WriteFile

SetConsoleOutputCP function

5/18/2021 • 2 minutes to read • Edit Online

Sets the output code page used by the console associated with the calling process. A console uses its output code page to translate the character values written by the various output functions into the images displayed in the console window.

Syntax

```
BOOL WINAPI SetConsoleOutputCP(
  _In_ UINT wCodePageID
);
```

Parameters

wCodePageID [in]
The identifier of the code page to set. For more information, see Remarks.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A code page maps 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages.

If the current font is a fixed-pitch Unicode font, **SetConsoleOutputCP** changes the mapping of the character values into the glyph set of the font, rather than loading a separate font each time it is called. This affects how extended characters (ASCII value greater than 127) are displayed in a console window. However, if the current font is a raster font, **SetConsoleOutputCP** does not affect how extended characters are displayed.

To find the code pages that are installed or supported by the operating system, use the EnumSystemCodePages function. The identifiers of the code pages available on the local computer are also stored in the registry under the following key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage
```

However, it is better to use EnumSystemCodePages to enumerate code pages because the registry can differ in different versions of Windows. To determine whether a particular code page is valid, use the IsValidCodePage function. To retrieve more information about a code page, including its name, use the **GetCPInfoEx** function. For a list of available code page identifiers, see Code Page Identifiers.

To determine a console's current output code page, use the **GetConsoleOutputCP** function. To set and retrieve a console's input code page, use the **SetConsoleCP** and **GetConsoleCP** functions.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Code Pages

Console Functions

GetConsoleCP

GetConsoleOutputCP

SetConsoleCP

SetConsoleScreenBufferInfoEx function

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets extended information about the specified console screen buffer.

Syntax

```
BOOL WINAPI SetConsoleScreenBufferInfoEx(
  _In_ HANDLE                      hConsoleOutput,
  _In_ PCONSOLE_SCREEN_BUFFER_INFOEX lpConsoleScreenBufferInfoEx
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

lpConsoleScreenBufferInfoEx [in]
A **CONSOLE_SCREEN_BUFFER_INFOEX** structure that contains the console screen buffer information.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

> **TIP**
> This API has a partial **virtual terminal** equivalent. **Cursor positioning buffer** and **text attributes** have specific sequence equivalents. The color table is not configurable, but **extended colors** are available beyond what is normally available through **console functions**. Popup attributes have no equivalent as popup menus are the responsibility of the command-line client application in the **virtual terminal** world. Finally, the size of the window and the full screen status are considered privileges owned by the user in the **virtual terminal** world and have no equivalent sequence.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

CONSOLE_SCREEN_BUFFER_INFOEX

GetConsoleScreenBufferInfoEx

SetConsoleScreenBufferSize function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Changes the size of the specified console screen buffer.

Syntax

```
BOOL WINAPI SetConsoleScreenBufferSize(
  _In_ HANDLE hConsoleOutput,
  _In_ COORD  dwSize
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

dwSize [in]
A **COORD** structure that specifies the new size of the console screen buffer, in character rows and columns. The specified width and height cannot be less than the width and height of the console screen buffer's window. The specified dimensions also cannot be less than the minimum size allowed by the system. This minimum depends on the current font size for the console (selected by the user) and the **SM_CXMIN** and **SM_CYMIN** values returned by the **GetSystemMetrics** function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Input Buffer

COORD

GetConsoleScreenBufferInfo

SetConsoleWindowInfo

SetConsoleTextAttribute function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets the attributes of characters written to the console screen buffer by the **WriteFile** or **WriteConsole** function, or echoed by the **ReadFile** or **ReadConsole** function. This function affects text written after the function call.

Syntax

```
BOOL WINAPI SetConsoleTextAttribute(
  _In_ HANDLE hConsoleOutput,
  _In_ WORD   wAttributes
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

wAttributes [in]
The character attributes.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

To determine the current color attributes of a screen buffer, call the **GetConsoleScreenBufferInfo** function.

TIP

This API has a **virtual terminal** equivalent in the **text formatting** sequences. *Virtual terminal sequences* are recommended for all new and ongoing development.

Examples

For an example, see Using the High-Level Input and Output Functions.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Screen Buffers

GetConsoleScreenBufferInfo

ReadConsole

ReadFile

WriteConsole

WriteFile

SetConsoleTitle function

8/3/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets the title for the current console window.

Syntax

```
BOOL WINAPI SetConsoleTitle(
  _In_ LPCTSTR lpConsoleTitle
);
```

Parameters

lpConsoleTitle [in]
The string to be displayed in the title bar of the console window. The total size must be less than 64K.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the process terminates, the system restores the original console title.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

TIP

This API has a **virtual terminal** equivalent in the **window title** sequences. *Virtual terminal sequences* are recommended for all new and ongoing development.

Examples

The following example shows how to retrieve and modify the console title.

```
#include <windows.h>
#include <tchar.h>
#include <conio.h>
#include <strsafe.h>

int main( void )
{
   TCHAR szOldTitle[MAX_PATH];
   TCHAR szNewTitle[MAX_PATH];

   // Save current console title.

   if( GetConsoleTitle(szOldTitle, MAX_PATH) )
   {
      // Build new console title string.

      StringCchPrintf(szNewTitle, MAX_PATH, TEXT("TEST: %s"), szOldTitle);

      // Set console title to new title
      if( !SetConsoleTitle(szNewTitle) )
      {
         _tprintf(TEXT("SetConsoleTitle failed (%d)\n"), GetLastError());
         return 1;
      }
      else
      {
         _tprintf(TEXT("SetConsoleTitle succeeded.\n"));
      }
   }
   return 0;
}
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**SetConsoleTitleW** (Unicode) and **SetConsoleTitleA** (ANSI)

See also

Console Functions

GetConsoleOriginalTitle

GetConsoleTitle

SetConsoleCP

SetConsoleOutputCP

SetConsoleWindowInfo function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets the current size and position of a console screen buffer's window.

Syntax

```
BOOL WINAPI SetConsoleWindowInfo(
  _In_       HANDLE     hConsoleOutput,
  _In_       BOOL       bAbsolute,
  _In_ const SMALL_RECT *lpConsoleWindow
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_READ** access right. For more information, see Console Buffer Security and Access Rights.

bAbsolute [in]
If this parameter is **TRUE**, the coordinates specify the new upper-left and lower-right corners of the window. If it is **FALSE**, the coordinates are relative to the current window-corner coordinates.

lpConsoleWindow [in]
A pointer to a **SMALL_RECT** structure that specifies the new upper-left and lower-right corners of the window.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The function fails if the specified window rectangle extends beyond the boundaries of the console screen buffer. This means that the **Top** and **Left** members of the *lpConsoleWindow* rectangle (or the calculated top and left coordinates, if *bAbsolute* is FALSE) cannot be less than zero. Similarly, the **Bottom** and **Right** members (or the calculated bottom and right coordinates) cannot be greater than (screen buffer height – 1) and (screen buffer width – 1), respectively. The function also fails if the **Right** member (or calculated right coordinate) is less than or equal to the **Left** member (or calculated left coordinate) or if the **Bottom** member (or calculated bottom coordinate) is less than or equal to the **Top** member (or calculated top coordinate).

For consoles with more than one screen buffer, changing the window location for one screen buffer does not affect the window locations of the other screen buffers.

To determine the current size and position of a screen buffer's window, use the **GetConsoleScreenBufferInfo** function. This function also returns the maximum size of the window, given the current screen buffer size, the current font size, and the screen size. The **GetLargestConsoleWindowSize** function returns the maximum window size given the current font and screen sizes, but it does not consider the size of the console screen buffer.

SetConsoleWindowInfo can be used to scroll the contents of the console screen buffer by shifting the position of the window rectangle without changing its size.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Examples

For an example, see Scrolling a Screen Buffer's Window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

GetConsoleScreenBufferInfo

GetLargestConsoleWindowSize

ScrollConsoleScreenBuffer

Scrolling the Screen Buffer

SMALL_RECT

SetCurrentConsoleFontEx function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Sets extended information about the current console font.

Syntax

```
BOOL WINAPI SetCurrentConsoleFontEx(
  _In_ HANDLE               hConsoleOutput,
  _In_ BOOL                 bMaximumWindow,
  _In_ PCONSOLE_FONT_INFOEX lpConsoleCurrentFontEx
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

bMaximumWindow [in]
If this parameter is **TRUE**, font information is set for the maximum window size. If this parameter is **FALSE**, font information is set for the current window size.

lpConsoleCurrentFontEx [in]
A pointer to a **CONSOLE_FONT_INFOEX** structure that contains the font information.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

TIP

This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems where the user is granted full control over this presentation option. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

CONSOLE_FONT_INFOEX

SetStdHandle function

9/16/2021 • 2 minutes to read • Edit Online

Sets the handle for the specified standard device (standard input, standard output, or standard error).

Syntax

```
BOOL WINAPI SetStdHandle(
  _In_ DWORD  nStdHandle,
  _In_ HANDLE hHandle
);
```

Parameters

nStdHandle [in]
The standard device for which the handle is to be set. This parameter can be one of the following values.

VALUE	MEANING
STD_INPUT_HANDLE `((DWORD)-10)`	The standard input device. Initially, this is the console input buffer, `CONIN$`.
STD_OUTPUT_HANDLE `((DWORD)-11)`	The standard output device. Initially, this is the active console screen buffer, `CONOUT$`.
STD_ERROR_HANDLE `((DWORD)-12)`	The standard error device. Initially, this is the active console screen buffer, `CONOUT$`.

> **NOTE**
> The values for these constants are unsigned numbers, but are defined in the header files as a cast from a signed number and take advantage of the C compiler rolling them over to just under the maximum 32-bit value. When interfacing with these handles in a language that does not parse the headers and is re-defining the constants, please be aware of this constraint. As an example, `((DWORD)-10)` is actually the unsigned number `4294967286`.

hHandle [in]
The handle for the standard device.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The standard handles of a process may have been redirected by a call to **SetStdHandle**, in which case **GetStdHandle** will return the redirected handle. If the standard handles have been redirected, you can specify the CONIN$ value in a call to the **CreateFile** function to get a handle to a console's input buffer. Similarly, you

can specify the CONOUT$ value to get a handle to the console's active screen buffer.

Examples

For an example, see Creating a Child Process with Redirected Input and Output.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ProcessEnv.h (via Winbase.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

Console Handles

CreateFile

GetStdHandle

WriteConsole function

5/18/2021 • 2 minutes to read • Edit Online

Writes a character string to a console screen buffer beginning at the current cursor location.

Syntax

```
BOOL WINAPI WriteConsole(
  _In_             HANDLE  hConsoleOutput,
  _In_       const VOID    *lpBuffer,
  _In_             DWORD   nNumberOfCharsToWrite,
  _Out_opt_        LPDWORD lpNumberOfCharsWritten,
  _Reserved_       LPVOID  lpReserved
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

lpBuffer [in]
A pointer to a buffer that contains characters to be written to the console screen buffer. This is expected to be an array of either `char` for `WriteConsoleA` or `wchar_t` for `WriteConsoleW`.

nNumberOfCharsToWrite [in]
The number of characters to be written. If the total size of the specified number of characters exceeds the available heap, the function fails with **ERROR_NOT_ENOUGH_MEMORY**.

lpNumberOfCharsWritten [out, optional]
A pointer to a variable that receives the number of characters actually written.

lpReserved Reserved; must be **NULL**.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteConsole** function writes characters to the console screen buffer at the current cursor position. The cursor position advances as characters are written. The **SetConsoleCursorPosition** function sets the current cursor position.

Characters are written using the foreground and background color attributes associated with the console screen buffer. The **SetConsoleTextAttribute** function changes these colors. To determine the current color attributes and the current cursor position, use **GetConsoleScreenBufferInfo**.

All of the input modes that affect the behavior of the **WriteFile** function have the same effect on **WriteConsole**. To retrieve and set the output modes of a console screen buffer, use the **GetConsoleMode** and

SetConsoleMode functions.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

WriteConsole fails if it is used with a standard handle that is redirected to a file. If an application processes multilingual output that can be redirected, determine whether the output handle is a console handle (one method is to call the **GetConsoleMode** function and check whether it succeeds). If the handle is a console handle, call **WriteConsole**. If the handle is not a console handle, the output is redirected and you should call **WriteFile** to perform the I/O. Be sure to prefix a Unicode plain text file with a byte order mark. For more information, see Using Byte Order Marks.

Although an application can use **WriteConsole** in ANSI mode to write ANSI characters, consoles do not support "ANSI escape" or "virtual terminal" sequences unless enabled. See **Console Virtual Terminal Sequences** for more information and for operating system version applicability.

When virtual terminal escape sequences are not enabled, console functions can provide equivalent functionality. For more information, see **SetCursorPos**, **SetConsoleTextAttribute**, and **GetConsoleCursorInfo**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**WriteConsoleW** (Unicode) and **WriteConsoleA** (ANSI)

See also

Console Functions

GetConsoleCursorInfo

GetConsoleMode

GetConsoleScreenBufferInfo

Input and Output Methods

ReadConsole

SetConsoleCP

SetConsoleCursorPosition

SetConsoleMode

SetConsoleOutputCP

SetConsoleTextAttribute

SetCursorPos

WriteFile

WriteConsoleInput function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Writes data directly to the console input buffer.

Syntax

```
BOOL WINAPI WriteConsoleInput(
  _In_        HANDLE       hConsoleInput,
  _In_  const INPUT_RECORD *lpBuffer,
  _In_        DWORD        nLength,
  _Out_       LPDWORD      lpNumberOfEventsWritten
);
```

Parameters

hConsoleInput [in]
A handle to the console input buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

lpBuffer [in]
A pointer to an array of **INPUT_RECORD** structures that contain data to be written to the input buffer.

nLength [in]
The number of input records to be written.

lpNumberOfEventsWritten [out]
A pointer to a variable that receives the number of input records actually written.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

WriteConsoleInput places input records into the input buffer behind any pending events in the buffer. The input buffer grows dynamically, if necessary, to hold as many events as are written.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions. Legacy consumers may also use the **chcp** or **mode con**

cp select= commands, but it is not recommended for new development.

> **TIP**
> This API is not recommended and does not have a **virtual terminal** equivalent. This decision intentionally aligns the Windows platform with other operating systems. This operation is considered the **wrong-way verb** for this buffer. Applications remoting via cross-platform utilities and transports like SSH may not work as expected if using this API.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**WriteConsoleInputW** (Unicode) and **WriteConsoleInputA** (ANSI)

See also

Console Functions

INPUT_RECORD

Low-Level Console Input Functions

MapVirtualKey

PeekConsoleInput

ReadConsoleInput

SetConsoleCP

SetConsoleOutputCP

VkKeyScan

WriteConsoleOutput function

5/18/2021 • 3 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Writes character and color attribute data to a specified rectangular block of character cells in a console screen buffer. The data to be written is taken from a correspondingly sized rectangular block at a specified location in the source buffer.

Syntax

```
BOOL WINAPI WriteConsoleOutput(
  _In_          HANDLE      hConsoleOutput,
  _In_    const CHAR_INFO   *lpBuffer,
  _In_          COORD       dwBufferSize,
  _In_          COORD       dwBufferCoord,
  _Inout_       PSMALL_RECT lpWriteRegion
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

lpBuffer [in]
The data to be written to the console screen buffer. This pointer is treated as the origin of a two-dimensional array of **CHAR_INFO** structures whose size is specified by the *dwBufferSize* parameter.

dwBufferSize [in]
The size of the buffer pointed to by the *lpBuffer* parameter, in character cells. The **X** member of the **COORD** structure is the number of columns; the **Y** member is the number of rows.

dwBufferCoord [in]
The coordinates of the upper-left cell in the buffer pointed to by the *lpBuffer* parameter. The **X** member of the **COORD** structure is the column, and the **Y** member is the row.

lpWriteRegion [in, out]
A pointer to a **SMALL_RECT** structure. On input, the structure members specify the upper-left and lower-right coordinates of the console screen buffer rectangle to write to. On output, the structure members specify the actual rectangle that was used.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

WriteConsoleOutput treats the source buffer and the destination screen buffer as two-dimensional arrays (columns and rows of character cells). The rectangle pointed to by the *lpWriteRegion* parameter specifies the size and location of the block to be written to in the console screen buffer. A rectangle of the same size is located with its upper-left cell at the coordinates of the *dwBufferCoord* parameter in the *lpBuffer* array. Data from the cells that are in the intersection of this rectangle and the source buffer rectangle (whose dimensions are specified by the *dwBufferSize* parameter) is written to the destination rectangle.

Cells in the destination rectangle whose corresponding source location are outside the boundaries of the source buffer rectangle are left unaffected by the write operation. In other words, these are the cells for which no data is available to be written.

Before **WriteConsoleOutput** returns, it sets the members of *lpWriteRegion* to the actual screen buffer rectangle affected by the write operation. This rectangle reflects the cells in the destination rectangle for which there existed a corresponding cell in the source buffer, because **WriteConsoleOutput** clips the dimensions of the destination rectangle to the boundaries of the console screen buffer.

If the rectangle specified by *lpWriteRegion* lies completely outside the boundaries of the console screen buffer, or if the corresponding rectangle is positioned completely outside the boundaries of the source buffer, no data is written. In this case, the function returns with the members of the structure pointed to by the *lpWriteRegion* parameter set such that the **Right** member is less than the **Left**, or the **Bottom** member is less than the **Top**. To determine the size of the console screen buffer, use the **GetConsoleScreenBufferInfo** function.

WriteConsoleOutput has no effect on the cursor position.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the **SetConsoleCP** or **SetConsoleOutputCP** functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

> **TIP**
> This API has a **virtual terminal** equivalent in the **text formatting** and **cursor positioning** sequences. Move the cursor to the location to insert, apply the formatting desired, and write out the text. *Virtual terminal sequences* are recommended for all new and ongoing development.

Examples

For an example, see Reading and Writing Blocks of Characters and Attributes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**WriteConsoleOutputW** (Unicode) and **WriteConsoleOutputA** (ANSI)

See also

Console Functions

CHAR_INFO

COORD

GetConsoleScreenBufferInfo

Low-Level Console Output Functions

ReadConsoleOutput

ReadConsoleOutputAttribute

ReadConsoleOutputCharacter

SetConsoleCP

SetConsoleOutputCP

SMALL_RECT

WriteConsoleOutputAttribute

WriteConsoleOutputCharacter

WriteConsoleOutputAttribute function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Copies a number of character attributes to consecutive cells of a console screen buffer, beginning at a specified location.

Syntax

```
BOOL WINAPI WriteConsoleOutputAttribute(
  _In_        HANDLE  hConsoleOutput,
  _In_  const WORD    *lpAttribute,
  _In_        DWORD   nLength,
  _In_        COORD   dwWriteCoord,
  _Out_       LPDWORD lpNumberOfAttrsWritten
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

lpAttribute [in]
The attributes to be used when writing to the console screen buffer. For more information, see Character Attributes.

nLength [in]
The number of screen buffer character cells to which the attributes will be copied.

dwWriteCoord [in]
A **COORD** structure that specifies the character coordinates of the first cell in the console screen buffer to which the attributes will be written.

lpNumberOfAttrsWritten [out]
A pointer to a variable that receives the number of attributes actually written to the console screen buffer.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the number of attributes to be written to extends beyond the end of the specified row in the console screen buffer, attributes are written to the next row. If the number of attributes to be written to extends beyond the end of the console screen buffer, the attributes are written up to the end of the console screen buffer.

The character values at the positions written to are not changed.

> **TIP**
> This API has a **virtual terminal** equivalent in the **text formatting** and **cursor positioning** sequences. Move the cursor to the location to insert, apply the formatting desired, and write out text to fill. There is no equivalent to apply color to an area without also emitting text. This decision intentionally aligns the Windows platform with other operating systems where the individual client application is expected to remember its own drawn state for further manipulation.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Console Functions

COORD

Low-Level Console Output Functions

ReadConsoleOutput

ReadConsoleOutputAttribute

ReadConsoleOutputCharacter

WriteConsoleOutput

WriteConsoleOutputCharacter

WriteConsoleOutputCharacter function

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Copies a number of characters to consecutive cells of a console screen buffer, beginning at a specified location.

Syntax

```
BOOL WINAPI WriteConsoleOutputCharacter(
  _In_  HANDLE  hConsoleOutput,
  _In_  LPCTSTR lpCharacter,
  _In_  DWORD   nLength,
  _In_  COORD   dwWriteCoord,
  _Out_ LPDWORD lpNumberOfCharsWritten
);
```

Parameters

hConsoleOutput [in]
A handle to the console screen buffer. The handle must have the **GENERIC_WRITE** access right. For more information, see Console Buffer Security and Access Rights.

lpCharacter [in]
The characters to be written to the console screen buffer.

nLength [in]
The number of characters to be written.

dwWriteCoord [in]
A **COORD** structure that specifies the character coordinates of the first cell in the console screen buffer to which characters will be written.

lpNumberOfCharsWritten [out]
A pointer to a variable that receives the number of characters actually written.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the number of characters to be written to extends beyond the end of the specified row in the console screen

buffer, characters are written to the next row. If the number of characters to be written to extends beyond the end of the console screen buffer, characters are written up to the end of the console screen buffer.

The attribute values at the positions written to are not changed.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the SetConsoleCP or SetConsoleOutputCP functions. Legacy consumers may also use the **chcp** or **mode con cp select=** commands, but it is not recommended for new development.

> **TIP**
> This API has a **virtual terminal** equivalent in the **text formatting** and **cursor positioning** sequences. Move the cursor to the location to insert, apply the formatting desired, and write out text to fill. There is no equivalent to emit text to an area without also applying the active color formatting. This decision intentionally aligns the Windows platform with other operating systems where the individual client application is expected to remember its own drawn state for further manipulation.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**WriteConsoleOutputCharacterW** (Unicode) and **WriteConsoleOutputCharacterA** (ANSI)

See also

Console Functions

COORD

Low-Level Console Output Functions

ReadConsoleOutput

ReadConsoleOutputAttribute

ReadConsoleOutputCharacter

SetConsoleCP

SetConsoleOutputCP

WriteConsoleOutput

WriteConsoleOutputAttribute

Console Structures

5/18/2021 • 2 minutes to read • Edit Online

The following structures are used to access a console.

- CHAR_INFO
- CONSOLE_CURSOR_INFO
- CONSOLE_FONT_INFO
- CONSOLE_FONT_INFOEX
- CONSOLE_HISTORY_INFO
- CONSOLE_READCONSOLE_CONTROL
- CONSOLE_SCREEN_BUFFER_INFO
- CONSOLE_SCREEN_BUFFER_INFOEX
- CONSOLE_SELECTION_INFO
- COORD
- FOCUS_EVENT_RECORD
- INPUT_RECORD
- KEY_EVENT_RECORD
- MENU_EVENT_RECORD
- MOUSE_EVENT_RECORD
- SMALL_RECT
- WINDOW_BUFFER_SIZE_RECORD

CONSOLE_HISTORY_INFO structure

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Contains information about the console history.

Syntax

```
typedef struct {
  UINT  cbSize;
  UINT  HistoryBufferSize;
  UINT  NumberOfHistoryBuffers;
  DWORD dwFlags;
} CONSOLE_HISTORY_INFO, *PCONSOLE_HISTORY_INFO;
```

Members

cbSize

The size of the structure, in bytes. Set this member to `sizeof(CONSOLE_HISTORY_INFO)`.

HistoryBufferSize

The number of commands kept in each history buffer.

NumberOfHistoryBuffers

The number of history buffers kept for this console process.

dwFlags

This parameter can be zero or the following value.

VALUE	MEANING
HISTORY_NO_DUP_FLAG 0x1	Duplicate entries will not be stored in the history buffer.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)

See also

GetConsoleHistoryInfo

SetConsoleHistoryInfo

CONSOLE_READCONSOLE_CONTROL structure

5/18/2021 • 2 minutes to read • Edit Online

Contains information for a console read operation.

Syntax

```
typedef struct _CONSOLE_READCONSOLE_CONTROL {
  ULONG nLength;
  ULONG nInitialChars;
  ULONG dwCtrlWakeupMask;
  ULONG dwControlKeyState;
} CONSOLE_READCONSOLE_CONTROL, *PCONSOLE_READCONSOLE_CONTROL;
```

Members

nLength
The size of the structure. Set this member to `sizeof(CONSOLE_READCONSOLE_CONTROL)`.

nInitialChars
The number of characters to skip (and thus preserve) before writing newly read input in the buffer passed to the ReadConsole function. This value must be less than the *nNumberOfCharsToRead* parameter of the **ReadConsole** function.

dwCtrlWakeupMask
A mask specifying which control characters between `0x00` and `0x1F` should be used to signal that the read is complete. Each bit corresponds to a character with the least significant bit corresponding to `0x00` or `NUL` and the most significant bit corresponding to `0x1F` or `US`. Multiple bits (control characters) can be specified.

dwControlKeyState
The state of the control keys. This member can be one or more of the following values.

VALUE	MEANING
CAPSLOCK_ON 0x0080	The CAPS LOCK light is on.
ENHANCED_KEY 0x0100	The key is enhanced. See remarks.
LEFT_ALT_PRESSED 0x0002	The left ALT key is pressed.
LEFT_CTRL_PRESSED 0x0008	The left CTRL key is pressed.
NUMLOCK_ON 0x0020	The NUM LOCK light is on.
RIGHT_ALT_PRESSED 0x0001	The right ALT key is pressed.
RIGHT_CTRL_PRESSED 0x0004	The right CTRL key is pressed.
SCROLLLOCK_ON 0x0040	The SCROLL LOCK light is on.

VALUE	MEANING
SHIFT_PRESSED 0x0010	The SHIFT key is pressed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi.h (via WinCon.h, include Windows.h)

See also

ReadConsole

CONSOLE_SELECTION_INFO structure

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Contains information for a console selection.

Syntax

```
typedef struct _CONSOLE_SELECTION_INFO {
  DWORD      dwFlags;
  COORD      dwSelectionAnchor;
  SMALL_RECT srSelection;
} CONSOLE_SELECTION_INFO, *PCONSOLE_SELECTION_INFO;
```

Members

dwFlags

The selection indicator. This member can be one or more of the following values.

VALUE	MEANING
CONSOLE_MOUSE_DOWN 0x0008	Mouse is down. The user is actively adjusting the selection rectangle with a mouse.
CONSOLE_MOUSE_SELECTION 0x0004	Selecting with the mouse. If off, the user is operating `conhost.exe` mark mode selection with the keyboard.
CONSOLE_NO_SELECTION 0x0000	No selection.
CONSOLE_SELECTION_IN_PROGRESS 0x0001	Selection has begun. If a mouse selection, this will typically not occur without the `CONSOLE_SELECTION_NOT_EMPTY` flag. If a keyboard selection, this may occur when mark mode has been entered but the user is still navigating to the initial position.
CONSOLE_SELECTION_NOT_EMPTY 0x0002	Selection rectangle not empty. The payload of *dwSelectionAnchor* and *srSelection* are valid.

dwSelectionAnchor

A COORD structure that specifies the selection anchor, in characters.

srSelection

A SMALL_RECT structure that specifies the selection rectangle.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	ConsoleApi3.h (via WinCon.h, include Windows.h)

See also

COORD

GetConsoleSelectionInfo

SMALL_RECT

CONSOLE_CURSOR_INFO structure

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Contains information about the console cursor.

Syntax

```
typedef struct _CONSOLE_CURSOR_INFO {
  DWORD dwSize;
  BOOL  bVisible;
} CONSOLE_CURSOR_INFO, *PCONSOLE_CURSOR_INFO;
```

Members

dwSize
The percentage of the character cell that is filled by the cursor. This value is between 1 and 100. The cursor appearance varies, ranging from completely filling the cell to showing up as a horizontal line at the bottom of the cell.

> **NOTE**
> While the **dwSize** value is normally between 1 and 100, under some circumstances a value outside of that range might be returned. For example, if **CursorSize** is set to 0 in the registry, the **dwSize** value returned would be 0.

bVisible
The visibility of the cursor. If the cursor is visible, this member is **TRUE**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinCon.h (include Windows.h)

See also

GetConsoleCursorInfo

CONSOLE_FONT_INFO structure

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Contains information for a console font.

Syntax

```
typedef struct _CONSOLE_FONT_INFO {
  DWORD nFont;
  COORD dwFontSize;
} CONSOLE_FONT_INFO, *PCONSOLE_FONT_INFO;
```

Members

nFont
The index of the font in the system's console font table.

dwFontSize
A COORD structure that contains the width and height of each character in the font, in logical units. The **X** member contains the width, while the **Y** member contains the height.

Remarks

To obtain the size of the font, pass the font index to the GetConsoleFontSize function.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	WinCon.h (include Windows.h)

See also

COORD

GetCurrentConsoleFont

CONSOLE_FONT_INFOEX structure

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Contains extended information for a console font.

Syntax

```
typedef struct _CONSOLE_FONT_INFOEX {
  ULONG cbSize;
  DWORD nFont;
  COORD dwFontSize;
  UINT  FontFamily;
  UINT  FontWeight;
  WCHAR FaceName[LF_FACESIZE];
} CONSOLE_FONT_INFOEX, *PCONSOLE_FONT_INFOEX;
```

Members

cbSize
The size of this structure, in bytes. This member must be set to `sizeof(CONSOLE_FONT_INFOEX)` before calling GetCurrentConsoleFontEx or it will fail.

nFont
The index of the font in the system's console font table.

dwFontSize
A COORD structure that contains the width and height of each character in the font, in logical units. The **X** member contains the width, while the **Y** member contains the height.

FontFamily
The font pitch and family. For information about the possible values for this member, see the description of the **tmPitchAndFamily** member of the TEXTMETRIC structure.

FontWeight
The font weight. The weight can range from 100 to 1000, in multiples of 100. For example, the normal weight is 400, while 700 is bold.

FaceName
The name of the typeface (such as Courier or Arial).

Remarks

To obtain the size of the font, pass the font index to the GetConsoleFontSize function.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	WinCon.h (include Windows.h)

See also

GetCurrentConsoleFontEx

CONSOLE_SCREEN_BUFFER_INFO structure

5/18/2021 • 2 minutes to read • Edit Online

Contains information about a console screen buffer.

Syntax

```
typedef struct _CONSOLE_SCREEN_BUFFER_INFO {
  COORD      dwSize;
  COORD      dwCursorPosition;
  WORD       wAttributes;
  SMALL_RECT srWindow;
  COORD      dwMaximumWindowSize;
} CONSOLE_SCREEN_BUFFER_INFO;
```

Members

dwSize
A **COORD** structure that contains the size of the console screen buffer, in character columns and rows.

dwCursorPosition
A **COORD** structure that contains the column and row coordinates of the cursor in the console screen buffer.

wAttributes
The attributes of the characters written to a screen buffer by the **WriteFile** and **WriteConsole** functions, or echoed to a screen buffer by the **ReadFile** and **ReadConsole** functions. For more information, see Character Attributes.

srWindow
A **SMALL_RECT** structure that contains the console screen buffer coordinates of the upper-left and lower-right corners of the display window.

dwMaximumWindowSize
A **COORD** structure that contains the maximum size of the console window, in character columns and rows, given the current screen buffer size and font and the screen size.

Examples

For an example, see Scrolling a Screen Buffer's Contents.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)

See also

COORD

GetConsoleScreenBufferInfo

ReadConsole

ReadFile

SMALL_RECT

WriteConsole

WriteFile

CONSOLE_SCREEN_BUFFER_INFOEX structure

5/18/2021 • 2 minutes to read • Edit Online

Contains extended information about a console screen buffer.

Syntax

```
typedef struct _CONSOLE_SCREEN_BUFFER_INFOEX {
  ULONG      cbSize;
  COORD      dwSize;
  COORD      dwCursorPosition;
  WORD       wAttributes;
  SMALL_RECT srWindow;
  COORD      dwMaximumWindowSize;
  WORD       wPopupAttributes;
  BOOL       bFullscreenSupported;
  COLORREF   ColorTable[16];
} CONSOLE_SCREEN_BUFFER_INFOEX, *PCONSOLE_SCREEN_BUFFER_INFOEX;
```

Members

cbSize

The size of this structure, in bytes.

dwSize

A **COORD** structure that contains the size of the console screen buffer, in character columns and rows.

dwCursorPosition

A **COORD** structure that contains the column and row coordinates of the cursor in the console screen buffer.

wAttributes

The attributes of the characters written to a screen buffer by the **WriteFile** and **WriteConsole** functions, or echoed to a screen buffer by the **ReadFile** and **ReadConsole** functions. For more information, see Character Attributes.

srWindow

A **SMALL_RECT** structure that contains the console screen buffer coordinates of the upper-left and lower-right corners of the display window.

dwMaximumWindowSize

A **COORD** structure that contains the maximum size of the console window, in character columns and rows, given the current screen buffer size and font and the screen size.

wPopupAttributes

The fill attribute for console pop-ups.

bFullscreenSupported

If this member is `TRUE`, full-screen mode is supported; otherwise, it is not. This will always be `FALSE` for systems after Windows Vista with the WDDM driver model as true direct VGA access to the monitor is no longer available.

ColorTable

An array of **COLORREF** values that describe the console's color settings.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ConsoleApi2.h (via WinCon.h, include Windows.h)

See also

COORD

GetConsoleScreenBufferInfoEx

SetConsoleScreenBufferInfoEx

SMALL_RECT

CHAR_INFO structure

9/16/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
>
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Specifies a Unicode or ANSI character and its attributes. This structure is used by console functions to read from and write to a console screen buffer.

Syntax

```
typedef struct _CHAR_INFO {
  union {
    WCHAR UnicodeChar;
    CHAR  AsciiChar;
  } Char;
  WORD  Attributes;
} CHAR_INFO, *PCHAR_INFO;
```

Members

Char

A union of the following members.

UnicodeChar

Unicode character of a screen buffer character cell.

AsciiChar

ANSI character of a screen buffer character cell.

Attributes

The character attributes. This member can be zero or any combination of the following values.

VALUE	MEANING
FOREGROUND_BLUE `0x0001`	Text color contains blue.
FOREGROUND_GREEN `0x0002`	Text color contains green.
FOREGROUND_RED `0x0004`	Text color contains red.
FOREGROUND_INTENSITY `0x0008`	Text color is intensified.
BACKGROUND_BLUE `0x0010`	Background color contains blue.

VALUE	MEANING
BACKGROUND_GREEN `0x0020`	Background color contains green.
BACKGROUND_RED `0x0040`	Background color contains red.
BACKGROUND_INTENSITY `0x0080`	Background color is intensified.
COMMON_LVB_LEADING_BYTE `0x0100`	Leading byte.
COMMON_LVB_TRAILING_BYTE `0x0200`	Trailing byte.
COMMON_LVB_GRID_HORIZONTAL `0x0400`	Top horizontal.
COMMON_LVB_GRID_LVERTICAL `0x0800`	Left vertical.
COMMON_LVB_GRID_RVERTICAL `0x1000`	Right vertical.
COMMON_LVB_REVERSE_VIDEO `0x4000`	Reverse foreground and background attribute.
COMMON_LVB_UNDERSCORE `0x8000`	Underscore.

Examples

For an example, see Scrolling a Screen Buffer's Contents.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinCon.h (include Windows.h)

See also

ReadConsoleOutput

ScrollConsoleScreenBuffer

WriteConsoleOutput

COORD structure

5/18/2021 • 2 minutes to read • Edit Online

Defines the coordinates of a character cell in a console screen buffer. The origin of the coordinate system (0,0) is at the top, left cell of the buffer.

Syntax

```
typedef struct _COORD {
  SHORT X;
  SHORT Y;
} COORD, *PCOORD;
```

Members

X

The horizontal coordinate or column value. The units depend on the function call.

Y

The vertical coordinate or row value. The units depend on the function call.

Examples

For an example, see Scrolling a Screen Buffer's Contents.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinConTypes.h (via WinCon.h, include Windows.h)

See also

CONSOLE_FONT_INFO

CONSOLE_SCREEN_BUFFER_INFO

CONSOLE_SELECTION_INFO

FillConsoleOutputAttribute

FillConsoleOutputCharacter

GetConsoleFontSize

GetLargestConsoleWindowSize

MOUSE_EVENT_RECORD

ReadConsoleOutput

ReadConsoleOutputAttribute

ReadConsoleOutputCharacter

ScrollConsoleScreenBuffer

SetConsoleCursorPosition

SetConsoleDisplayMode

SetConsoleScreenBufferSize

WINDOW_BUFFER_SIZE_RECORD

WriteConsoleOutput

WriteConsoleOutputAttribute

WriteConsoleOutputCharacter

SMALL_RECT structure

5/18/2021 • 2 minutes to read • Edit Online

Defines the coordinates of the upper left and lower right corners of a rectangle.

Syntax

```
typedef struct _SMALL_RECT {
  SHORT Left;
  SHORT Top;
  SHORT Right;
  SHORT Bottom;
} SMALL_RECT;
```

Members

Left
The x-coordinate of the upper left corner of the rectangle.

Top
The y-coordinate of the upper left corner of the rectangle.

Right
The x-coordinate of the lower right corner of the rectangle.

Bottom
The y-coordinate of the lower right corner of the rectangle.

Remarks

This structure is used by console functions to specify rectangular areas of console screen buffers, where the coordinates specify the rows and columns of screen-buffer character cells.

Examples

For an example, see Scrolling a Screen Buffer's Contents.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinConTypes.h (via WinCon.h, include Windows.h)

See also

RECT

INPUT_RECORD structure

5/18/2021 • 2 minutes to read • Edit Online

Describes an input event in the console input buffer. These records can be read from the input buffer by using the ReadConsoleInput or PeekConsoleInput function, or written to the input buffer by using the WriteConsoleInput function.

Syntax

```
typedef struct _INPUT_RECORD {
  WORD  EventType;
  union {
    KEY_EVENT_RECORD          KeyEvent;
    MOUSE_EVENT_RECORD        MouseEvent;
    WINDOW_BUFFER_SIZE_RECORD WindowBufferSizeEvent;
    MENU_EVENT_RECORD         MenuEvent;
    FOCUS_EVENT_RECORD        FocusEvent;
  } Event;
} INPUT_RECORD;
```

Members

EventType
A handle to the type of input event and the event record stored in the **Event** member.

This member can be one of the following values.

VALUE	MEANING
FOCUS_EVENT 0x0010	The **Event** member contains a FOCUS_EVENT_RECORD structure. These events are used internally and should be ignored.
KEY_EVENT 0x0001	The **Event** member contains a KEY_EVENT_RECORD structure with information about a keyboard event.
MENU_EVENT 0x0008	The **Event** member contains a MENU_EVENT_RECORD structure. These events are used internally and should be ignored.
MOUSE_EVENT 0x0002	The **Event** member contains a MOUSE_EVENT_RECORD structure with information about a mouse movement or button press event.
WINDOW_BUFFER_SIZE_EVENT 0x0004	The **Event** member contains a WINDOW_BUFFER_SIZE_RECORD structure with information about the new size of the console screen buffer.

Event
The event information. The format of this member depends on the event type specified by the **EventType** member.

Examples

For an example, see Reading Input Buffer Events.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinConTypes.h (via WinCon.h, include Windows.h)

See also

FOCUS_EVENT_RECORD

KEY_EVENT_RECORD

MENU_EVENT_RECORD

MOUSE_EVENT_RECORD

PeekConsoleInput

ReadConsoleInput

WriteConsoleInput

KEY_EVENT_RECORD structure

5/18/2021 • 2 minutes to read • Edit Online

Describes a keyboard input event in a console **INPUT_RECORD** structure.

Syntax

```
typedef struct _KEY_EVENT_RECORD {
  BOOL  bKeyDown;
  WORD  wRepeatCount;
  WORD  wVirtualKeyCode;
  WORD  wVirtualScanCode;
  union {
    WCHAR UnicodeChar;
    CHAR  AsciiChar;
  } uChar;
  DWORD dwControlKeyState;
} KEY_EVENT_RECORD;
```

Members

bKeyDown
If the key is pressed, this member is **TRUE**. Otherwise, this member is **FALSE** (the key is released).

wRepeatCount
The repeat count, which indicates that a key is being held down. For example, when a key is held down, you might get five events with this member equal to 1, one event with this member equal to 5, or multiple events with this member greater than or equal to 1.

wVirtualKeyCode
A virtual-key code that identifies the given key in a device-independent manner.

wVirtualScanCode
The virtual scan code of the given key that represents the device-dependent value generated by the keyboard hardware.

uChar
A union of the following members.

UnicodeChar
Translated Unicode character.

AsciiChar
Translated ASCII character.

dwControlKeyState
The state of the control keys. This member can be one or more of the following values.

VALUE	MEANING
CAPSLOCK_ON 0x0080	The CAPS LOCK light is on.
ENHANCED_KEY 0x0100	The key is enhanced. See remarks.

VALUE	MEANING
LEFT_ALT_PRESSED 0x0002	The left ALT key is pressed.
LEFT_CTRL_PRESSED 0x0008	The left CTRL key is pressed.
NUMLOCK_ON 0x0020	The NUM LOCK light is on.
RIGHT_ALT_PRESSED 0x0001	The right ALT key is pressed.
RIGHT_CTRL_PRESSED 0x0004	The right CTRL key is pressed.
SCROLLLOCK_ON 0x0040	The SCROLL LOCK light is on.
SHIFT_PRESSED 0x0010	The SHIFT key is pressed.

Remarks

Enhanced keys for the IBM® 101- and 102-key keyboards are the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and direction keys in the clusters to the left of the keypad; and the divide (/) and ENTER keys in the keypad.

Keyboard input events are generated when any key, including control keys, is pressed or released. However, the ALT key when pressed and released without combining with another character, has special meaning to the system and is not passed through to the application. Also, the CTRL+C key combination is not passed through if the input handle is in processed mode (**ENABLE_PROCESSED_INPUT**).

Examples

For an example, see Reading Input Buffer Events.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinConTypes.h (via WinCon.h, include Windows.h)

See also

PeekConsoleInput

ReadConsoleInput

WriteConsoleInput

INPUT_RECORD

MENU_EVENT_RECORD structure

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Describes a menu event in a console **INPUT_RECORD** structure. These events are used internally and should be ignored.

Syntax

```
typedef struct _MENU_EVENT_RECORD {
  UINT dwCommandId;
} MENU_EVENT_RECORD, *PMENU_EVENT_RECORD;
```

Members

dwCommandId
Reserved.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinConTypes.h (via WinCon.h, include Windows.h)

See also

INPUT_RECORD

MOUSE_EVENT_RECORD structure

5/18/2021 • 2 minutes to read • Edit Online

> **IMPORTANT**
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

Describes a mouse input event in a console INPUT_RECORD structure.

Syntax

```
typedef struct _MOUSE_EVENT_RECORD {
  COORD dwMousePosition;
  DWORD dwButtonState;
  DWORD dwControlKeyState;
  DWORD dwEventFlags;
} MOUSE_EVENT_RECORD;
```

Members

dwMousePosition

A COORD structure that contains the location of the cursor, in terms of the console screen buffer's character-cell coordinates.

dwButtonState

The status of the mouse buttons. The least significant bit corresponds to the leftmost mouse button. The next least significant bit corresponds to the rightmost mouse button. The next bit indicates the next-to-leftmost mouse button. The bits then correspond left to right to the mouse buttons. A bit is 1 if the button was pressed.

The following constants are defined for the first five mouse buttons.

VALUE	MEANING
FROM_LEFT_1ST_BUTTON_PRESSED 0x0001	The leftmost mouse button.
FROM_LEFT_2ND_BUTTON_PRESSED 0x0004	The second button fom the left.
FROM_LEFT_3RD_BUTTON_PRESSED 0x0008	The third button from the left.
FROM_LEFT_4TH_BUTTON_PRESSED 0x0010	The fourth button from the left.
RIGHTMOST_BUTTON_PRESSED 0x0002	The rightmost mouse button.

dwControlKeyState

The state of the control keys. This member can be one or more of the following values.

VALUE	MEANING
CAPSLOCK_ON 0x0080	The CAPS LOCK light is on.
ENHANCED_KEY 0x0100	The key is enhanced. See remarks.
LEFT_ALT_PRESSED 0x0002	The left ALT key is pressed.
LEFT_CTRL_PRESSED 0x0008	The left CTRL key is pressed.
NUMLOCK_ON 0x0020	The NUM LOCK light is on.
RIGHT_ALT_PRESSED 0x0001	The right ALT key is pressed.
RIGHT_CTRL_PRESSED 0x0004	The right CTRL key is pressed.
SCROLLLOCK_ON 0x0040	The SCROLL LOCK light is on.
SHIFT_PRESSED 0x0010	The SHIFT key is pressed.

dwEventFlags

The type of mouse event. If this value is zero, it indicates a mouse button being pressed or released. Otherwise, this member is one of the following values.

VALUE	MEANING
DOUBLE_CLICK 0x0002	The second click (button press) of a double-click occurred. The first click is returned as a regular button-press event.
MOUSE_HWHEELED 0x0008	The horizontal mouse wheel was moved. If the high word of the **dwButtonState** member contains a positive value, the wheel was rotated to the right. Otherwise, the wheel was rotated to the left.
MOUSE_MOVED 0x0001	A change in mouse position occurred.
MOUSE_WHEELED 0x0004	The vertical mouse wheel was moved. If the high word of the **dwButtonState** member contains a positive value, the wheel was rotated forward, away from the user. Otherwise, the wheel was rotated backward, toward the user.

Remarks

Mouse events are placed in the input buffer when the console is in mode (**ENABLE_MOUSE_INPUT**).

Mouse events are generated whenever the user moves the mouse, or presses or releases one of the mouse buttons. Mouse events are placed in a console's input buffer only when the console group has the keyboard focus and the cursor is within the borders of the console's window.

Examples

For an example, see Reading Input Buffer Events.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinConTypes.h (via WinCon.h, include Windows.h)

See also

COORD

INPUT_RECORD

PeekConsoleInput

ReadConsoleInput

WriteConsoleInput

FOCUS_EVENT_RECORD structure

5/18/2021 • 2 minutes to read • Edit Online

Describes a focus event in a console INPUT_RECORD structure. These events are used internally and should be ignored.

Syntax

```
typedef struct _FOCUS_EVENT_RECORD {
  BOOL bSetFocus;
} FOCUS_EVENT_RECORD;
```

Members

bSetFocus
Reserved.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinConTypes.h (via WinCon.h, include Windows.h)

See also

INPUT_RECORD

WINDOW_BUFFER_SIZE_RECORD structure

5/18/2021 • 2 minutes to read • Edit Online

Describes a change in the size of the console screen buffer.

Syntax

```
typedef struct _WINDOW_BUFFER_SIZE_RECORD {
  COORD dwSize;
} WINDOW_BUFFER_SIZE_RECORD;
```

Members

dwSize
A **COORD** structure that contains the size of the console screen buffer, in character cell columns and rows.

Remarks

Buffer size events are placed in the input buffer when the console is in window-aware mode (**ENABLE_WINDOW_INPUT**).

Examples

For an example, see Reading Input Buffer Events.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	WinConTypes.h (via WinCon.h, include Windows.h)

See also

COORD

INPUT_RECORD

ReadConsoleInput

Console WinEvents

5/18/2021 • 2 minutes to read • Edit Online

IMPORTANT

WinEvents are part of the legacy **Microsoft Active Accessibility** framework. Development using these events is strongly discouraged in favor of the **Microsoft UI Automation** framework which provides a more robust and comprehensive suite of interfaces for accessibility and automation applications to interact with the console.

WARNING

Registering for these events is a global activity and will significantly impact performance of all command-line applications running on a system at the same time, including services and background utilities. The **Microsoft UI Automation** framework is console session specific and overcomes this limitation.

The following event constants are used in the *event* parameter of the *WinEventProc* callback function. For more information, see WinEvents.

CONSTANT/VALUE	DESCRIPTION
EVENT_CONSOLE_CARET 0x4001	The console caret has moved. The *idObject* parameter is one or more of the following values: **CONSOLE_CARET_SELECTION** or **CONSOLE_CARET_VISIBLE**. The *idChild* parameter is a **COORD** structure that specifies the cursor's current position.
EVENT_CONSOLE_END_APPLICATION 0x4007	A console process has exited. The *idObject* parameter contains the process identifier of the terminated process.
EVENT_CONSOLE_LAYOUT 0x4005	The console layout has changed.
EVENT_CONSOLE_START_APPLICATION 0x4006	A new console process has started. The *idObject* parameter contains the process identifier of the newly created process. If the application is a 16-bit application, the *idChild* parameter is **CONSOLE_APPLICATION_16BIT** and *idObject* is the process identifier of the NTVDM session associated with the console.
EVENT_CONSOLE_UPDATE_REGION 0x4002	More than one character has changed. The *idObject* parameter is a **COORD** structure that specifies the start of the changed region. The *idChild* parameter is a **COORD** structure that specifies the end of the changed region.
EVENT_CONSOLE_UPDATE_SCROLL 0x4004	The console has scrolled. The *idObject* parameter is the horizontal distance the console has scrolled. The *idChild* parameter is the vertical distance the console has scrolled.

CONSTANT/VALUE	DESCRIPTION
EVENT_CONSOLE_UPDATE_SIMPLE 0x4003	A single character has changed. The *idObject* parameter is a **COORD** structure that specifies the character that has changed. The *idChild* parameter specifies the character in the low word and the **character attributes** in the high word.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h