

Windows Console and Terminal Definitions

Article • 09/21/2022

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.

Windows Console and Terminal Ecosystem Roadmap

Article • 09/21/2022

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.

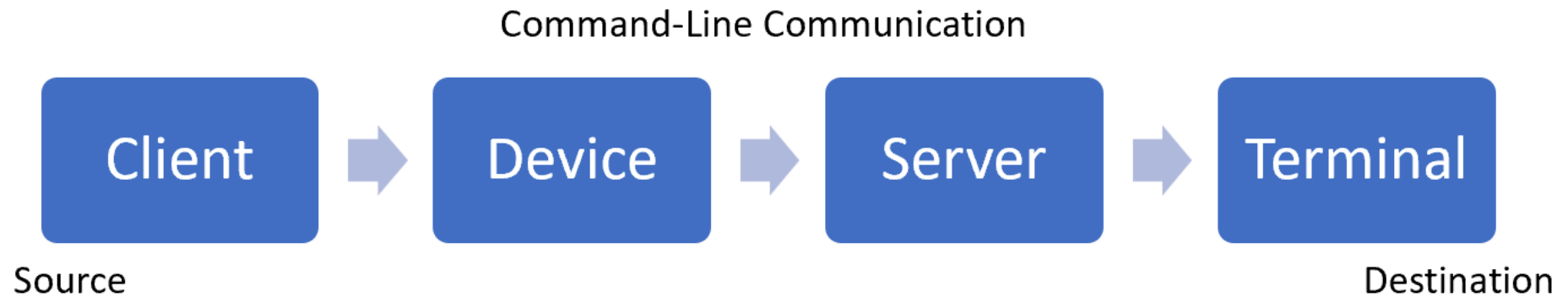
The focus of the current console/terminal era at Microsoft is to bring a first-class terminal experience directly to developers on the Windows platform and to [phase out](#) classic Windows Console APIs, replacing them with [virtual terminal sequences](#) utilizing [pseudoconsole](#). **Windows Terminal** 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](https://github.com/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.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)  | [Get help at Microsoft Q&A](#)

Classic Console APIs versus Virtual Terminal Sequences

Article • 09/21/2022

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](https://github.com/microsoft/terminal/issues/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](#).

Feedback

Was this page helpful?



Yes



No

[Provide product feedback](#)  | [Get help at Microsoft Q&A](#)

About Character Mode Applications

Article • 10/30/2020

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](#)
-

Consoles

Article • 10/30/2020

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](#)
-

Pseudoconsoles

Article • 09/21/2022

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 \(ConPTY\)](#) [↗](#).

Feedback