Niksa's explanations

Sometimes @miniksa will write a big, long explanatory comment in an issue thread that turns out to be a decent bit of reference material. This document serves as a storage point for those posts.

- Why do we avoid changing CMD.exe?
- Why is typing-to-screen performance better than every other app?
- How are the Windows graphics/messaging stack assembled?
- Output Processing between "Far East" and "Western"
- Why do we not backport things?
- Why can't we have mixed elevated and non-elevated tabs in the Terminal?
- What's the difference between a shell and a terminal?

Why do we avoid changing CMD.exe?

`setlocal` doesn't behave the same way as an environment variable. It's a thing that would have to be put in at the top of the batch script that is `somefile.cmd` as one of its first commands to adjust the way that one specific batch file is processed by the `cmd.exe` engine. That's probably not suitable for your needs, but that's the way we have to go.

I don't think anyone is disagreeing with you, @mikemaccana, that this would be a five minute development change to read that environment variable and change the behavior of `cmd.exe`. It absolutely would be a tiny development time.

It's just that from our experience, we know there's going to be a 3-24 month bug tail here where we get massive investigation callbacks by some billion dollar enterprise customer who for whatever reason was already using the environment variable we pick for another purpose. Their script that they give their rank-and-file folks will tell them to press Ctrl+C at some point in the batch script to do whatever happens, it will do something different, those people will notice the script doesn't match the computer anymore. They will then halt the production line and tell their supervisor. The supervisor tells some director. Their director comes screaming at their Microsoft enterprise support contract person that we've introduced a change to the OS that is costing them millions if not billions of dollars in shipments per month. Our directors at Microsoft then come bashing down our doors angry with us and make us fix it ASAP or revert it, we don't get to go home at 5pm to our families or friends because we're fixing it, we get stressed the heck out, we have to spin up servicing potentially for already shipped operating systems which is expensive and headache-causing... etc.

We can see this story coming a million miles away because it has happened before with other 'tiny' change we've been asked to make to `cmd.exe` in the past few years.

I would just ask you to understand that `cmd.exe` is very, very much in a maintenance mode and I just want to set expectations here. We maintain it, yes.

We have a renewed interest in command-line development, yes. But our focuses are revolving around improving the terminal and platform itself and bringing modern, supported shells to be the best they can be on Windows. Paul will put this on the backlog of things that people want in `cmd.exe`, yes. But it will sink to the bottom of the backlog because changing `cmd.exe` is our worst nightmare as its compatibility story is among the heaviest of any piece of the operating system.

I would highly recommend that Gulp convert to using PowerShell scripts and that if such an issue exists with PowerShell, that we get their modern, supported, and better-engineered platform to support the scenario. I don't want you to sit around waiting for `cmd.exe` to change this because it's really not going to happen faster than that script could be converted to `ps1` and it fixed in PowerShell Core (if that's even a problem in that world.)

Original Source: https://github.com/microsoft/terminal/issues/217#issuecomment-404240443

Why is typing-to-screen performance better than every other app?

I really do not mind when someone comes by and decides to tell us that we're doing a good job at something. We hear so many complaints every day that a post like this is a breath of fresh air. Thanks for your thanks!

Also, I'm happy to discuss this with you until you're utterly sick of reading it. Please ask any follow-ons you want. I thrive on blathering about my work. :P

If I had to take an educated guess as to what is making us faster than pretty much any other application on Windows at putting your text on the screen... I would say it is because that is literally our only job! Also probably because we are using darn near the oldest and lowest level APIs that Windows has to accomplish this work.

Pretty much everything else you've listed has some sort of layer or framework involved, or many, many layers and frameworks, when you start talking about Electron and JavaScript. We don't.

We have one bare, super un-special window with no additional controls attached to it. We get our keys fed into us from just barely above the kernel given that we're processing them from window messages and not from some sort of eventing framework common to pretty much any other more complicated UI framework than ours (WPF, WinForms, UWP, Electron). And we dump our text straight onto the window surface using GDI's PolyTextOut with no frills.

Even `notepad.exe` has multiple controls on its window at the very least and is probably (I haven't looked) using some sort of library framework in the edit control to figure out its text layout (which probably is using another library framework for internationalization support...)

Of course this also means that we have trade offs. We don't support fully international text like pretty much every other application will. RTL? No go zone right now. Surrogate pairs and emoji? We're getting there but not there yet. Indic scripts? Nope.

Why are we like this? For one, `conhost.exe` is old as dirt. It has to use the bare metal bottom layer of everything because it was created before most of those other frameworks were created. And also it maintains as low/bottom level as possible because it is pretty much the first thing that one needs to bring up when bringing up a new operating system edition or device before you have all the nice things like frameworks or what those frameworks require to operate. Also it's written in C/C++ which is about as low and bare metal as we can get.

Will this UI enhancement come to other apps on Windows? Almost certainly not. They have too much going on which is both a good and a bad thing. I'm jealous of their ability to just call one method and layout text in an uncomplicated manner in any language without manually calculating pixels or caring about what styles apply to their font. But my manual pixel calculations, dirty region math, scroll region madness, and more makes it so we go faster than them. I'm also jealous that when someone says "hey can you add a status bar to the bottom of your window" that they can pretty much click and drag that into place with their UI Framework and it will just work where as for us, it's been a backlog item forever and gives me heartburn to think about implementing.

Will we try to keep it from regressing? Yes! Right now it's sort of a manual process. We identify that something is getting slow and then we go haul out WPR and start taking traces. We stare down the hot paths and try to reason out what is going on and then improve them. For instance, in the last cycle or two, we focused on heap allocations as a major area where we could improve our end-to-end performance, changing a ton of our code to use stack-constructed iterator-like facades over the underlying request buffer instead of translating and allocating it into a new heap space for each level of processing.

As an aside, @bitcrazed wants us to automate performance tests in some conhost specific way, but I haven't quite figured out a controlled environment to do this in yet. The Windows Engineering System runs performance tests each night that give us a coarse grained way of knowing if we messed something up for the whole operating system, and they technically offer a fine grained way for us to insert our own performance tests... but I just haven't got around to that yet. If you have an idea for a way for us to do this in an automated fashion, I'm all ears.

If there's anything else you'd like to know, let me know. I could go on all day. I deleted like 15 tangents from this reply before posting it....

Original Source: https://github.com/microsoft/terminal/issues/327#issuecomment-447391705

How are the Windows graphics/messaging stack assembled?

@stakx, I am referring to USER32 and GDI32.

I'll give you a cursory overview of what I know off the top of my head without spending hours confirming the details. As such, some of this is subject to handwaving and could be mildly incorrect but is probably in the right direction. Consider every statement to be my personal knowledge on how the world works and subject to opinion or error.

For the graphics part of the pipeline (GDI32), the user-mode portions of GDI are pretty far down. The app calls GDI32, some work is done in that DLL on the user-mode side, then a kernel call jumps over to the kernel and drawing occurs.

The portion that you're thinking of regarding "silently converted to sit on top of other stuff" is probably that once we hit the kernel calls, a bunch of the kernel GDI stuff tends to be re-platformed on top of the same stuff as DirectX when it is actually handled by the NVIDIA/AMD/Intel/etc. graphics driver and the GPU at the bottom of the stack. I think this happened with the graphics driver re-architecture that came as a part of WDDM for Windows Vista. There's a document out there somewhere about what calls are still really fast in GDI and which are slower as a result of the re-platforming. Last time I found that document and checked, we were using the fast ones.

On top of GDI, I believe there are things like Common Controls or comctl32.dll which provided folks reusable sets of buttons and elements to make their UIs before we had nicer declarative frameworks. We don't use those in the console really (except in the property sheet off the right click menu).

As for DirectWrite and D2D and D3D and DXGI themselves, they're a separate set of commands and paths that are completely off to the side from GDI at all both in user and kernel mode. They're not really related other than that there's some interoperability provisions between the two. Most of our other UI frameworks tend to be built on top of the DirectX stack though. XAML is for sure. I think WPF is. Not sure about WinForms. And I believe the composition stack and the window manager are using DirectX as well.

As for the input/interaction part of the pipeline (USER32), I tend to find most other newer things (at least for desktop PCs) are built on top of what is already there. USER32's major concept is windows and window handles and everything is sent to a window handle. As long as you're on a desktop machine (or a laptop or whatever... I mean a classic-style Windows-powered machine), there's a window handle involved and messages floating around and that means we're talking USER32.

The window message queue is just a straight up FIFO (more or less) of whatever input has occurred relevant to that window while it's in the foreground + whatever has been sent to the window by other components in the system.

The newer technologies and the frameworks like XAML and WPF and WinForms

tend to receive the messages from the window message queue one way or another and process them and turn them into event callbacks to various objects that they've provisioned within their world.

However, the newer technologies that also work on other non-desktop platforms like XAML tend to have the ability to process stuff off of a completely different non-USER32 stack as well. There's a separate parallel stack to USER32 with all of our new innovations and realizations on how input and interaction should occur that doesn't exactly deal with classic messaging queues and window handles the same way. This is the whole Core* family of things like CoreWindow and CoreMessaging. They also have a different concept of "what is a user" that isn't so centric around your butt in rolling chair in front of a screen with a keyboard and mouse on the desk.

Now, if you're on XAML or one of the other Frameworks... all this intricacy is handled for you. XAML figures out how to draw on DirectX for you and negotiates with the compositor and window manager for cool effects on your behalf. It figures out whether to get your input events from USER32 or Core* or whatever transparently depending on your platform and the input stacks can handle pen, touch, keyboard, mouse, and so on in a unified manner. It has provisions inside it embedded to do all the sorts of globalization, accessibility, input interaction, etc. stuff that make your life easy. But you could choose to go directly to the low-level and handle it yourself or skip handling what you don't care about.

The trick is that GDI32 and USER32 were designed for a limited world with a limited set of commands. Desktop PCs were the only thing that existed, single user at the keyboard and mouse, simple graphics output to a VGA monitor. So using them directly at the "low level" like conhost does is pretty easy. The new platforms could be used at the "low level" but they're orders of magnitude more complicated because they now account for everything that has happened with personal computing in 20+ years like different form factors, multiple active users, multiple graphics adapters, and on and on and on and on. So you tend to use a framework when using the new stuff so your head doesn't explode. They handle it for you, but they handle more than they ever did before so they're slower to some degree.

So are GDI32 and USER32 "lower" than the new stuff? Sort of. Can you get that low with the newer stuff? Mostly yes, but you probably shouldn't and don't want to. Does new live on top of old or is old replatformed on the new? Sometimes and/or partially. Basically... it's like the answer to anything software... "it's an unmitigated disaster and if we all stepped back a moment, we should be astounded that it works at all." :P

Anyway, that's enough ramble for one morning. Hopefully that somewhat answered your questions and gave you a bit more insight.

Original Source: https://github.com/microsoft/terminal/issues/327#issuecomment-447926388

Output Processing between “Far East” and “Western”

```
if (WI_IsFlagSet(CharType, C1_CNTRL))
```

In short, this is probably fine to fix.

However, I would personally feed a few characters through `WriteCharsLegacy` under the debugger and assert that your theory is correct first (that multiple flags coming back are what the problem is) before making the change.

I am mildly terrified, less than Dustin, because it is freaking `WriteCharsLegacy` which is the spawn of hell and I fear some sort of regression in it.

In long, why is it fine to fix?

For reference, this particular segment of code https://github.com/microsoft/terminal/blob/9b92986b49bed8cc41f
L539 appears to only be used when the codepoint is $<$ 0x20 or $==$ 0x7F
https://github.com/microsoft/terminal/blob/9b92986b49bed8cc41fde4d6ef080921c41e6d9e/src/host/_stream.cp
and ENABLE_PROCESSED_OUTPUT is off. https://github.com/microsoft/terminal/blob/9b92986b49bed8cc

I looked back at the console v1 code and this particular section had a divergence for “Western” countries and “Far East” countries (a geopolitically-charged term, but what it was, nonetheless.)

For “Western” countries, we would unconditionally run all the characters through `MultiByteToWideChar` with `MB_USEGLYPHCHARS` without the `C1_CNTRL` test and move the result into the buffer.

For “Eastern” countries, we did the `C1_CNTRL` test and then if true, we would run through `MultiByteToWideChar` with `MB_USEGLYPHCHARS`. Otherwise, we would just move the original character into the buffer and call it a day.

Note in both of these, there is a little bit of indirection before `MultiByteToWideChar` is called through some other helper methods like `ConvertOutputToUnicode`, but that’s the effective conversion point, as far as I can tell. And that’s where the control characters would turn into acceptable low ASCII symbols.

When we took over the console codebase, this variation between “Western” and “Eastern” countries was especially painful because `conhost.exe` would choose which one it was in based on the `Codepage for Non-Unicode Applications` set in the Control Panel’s Regional > Administrative panel and it could only be changed with a reboot. It wouldn’t even change properly when you `chcp` to a different codepage. Heck, `chcp` would deny you from switching into many codepages. There was a block in place to prevent going to an “Eastern” codepage if you booted up in a “Western” codepage. There was also a block preventing you from going between “Eastern” codepages, if I recall correctly.

In modernizing, I decided a few things: 1. What’s good for the “Far East” should be good for the rest of the world. CJK languages that encompassed the “Far East” code have to be able to handle “Western” text as well even if the reverse wasn’t true. 2. We need to scrub all usages of “Far East” from the

code. Someone already started that and replaced them with "East Asia" except then they left behind the shorthand of "FE" prefixing dozens of functions which made it hard to follow the code. It took us months to realize "FE" and "East Asia" were the same thing. 3. It's obnoxious that the way this was handled was to literally double-define every output function in the code base to have two definitions, compile them both into the conhost, then choose to run down the SB_ versions or the FE_ versions depending on the startup Non-Unicode codepage. It was a massive pile of complex pre-compilation `#ifdef` and `#else`s that would sometimes surround individual lines in the function bodies. Gross. 4. The fact that the FE_ versions of the functions were way slower than the SB_ ones was unacceptable even for the same output of Latin-character text. 5. Anyone should be free to switch between any codepage they want at any time and restricting it based on a value from OS startup or region/locale is not acceptable in the modern world. 6. I concluded by all of the above that I was going to tank/delete/remove the SB_ versions of everything and force the entire world to use the FE_ versions as truth. I would fix the FE_ versions to handle everything correctly, I would fix the performance characteristics of the FE_ versions so they were only slower when things were legitimately more complicated and never otherwise, I would banish all usage of "Far East", "East Asia", "FE_", and "SB_" from the codebase, and codepages would be freely switchable. 7. Oh. Also, the conhost used to rewrite its entire backing buffer into whatever your current codepage was whenever you switched codepages. I changed that to always hold it as UTF-16.

Now, after that backstory. This is where the problem comes in. It looks like the code you're pointing to that didn't check flags and instead checked direct equality... is the way that it was ALWAYS done for the "Eastern" copy of the code. So it was ALWAYS broken for the "Eastern" codepages and country variants of the OS.

I don't know why the "Eastern" copy was checking `C1_CNTRL` at all in the first place. There is no documentation. I presume it has to do with Shift-JIS or GB-2312 or Unified Hangul or something having a conflict $< 0x20$ || $== 0x7F$. Or alternatively, it's because someone wrote the code naively thinking it was a good idea in a hurry and never tested it. Very possible and even probable.

Presuming CJK codepages have no conflict in this range for their DBCS codepages... we could probably remove the check with `GetStringTypeW` entirely and always run everything through `ConvertOutputToUnicode`. More risky than just the flag test change... but theoretically an option as well.

Original Source: https://github.com/microsoft/terminal/issues/166#issuecomment-510953359

Why do we not backport things?

Someone has to prove that this is costing millions to billions of dollars of lost productivity or revenue to outweigh the risks of shipping the fix to hundreds of

millions of Windows machines and potentially breaking something.

Our team generally finds it pretty hard to prove that against the developer audience given that they're only a small portion of the total installed market of Windows machines.

Our only backport successes really come from corporations with massive addressable market (like OEMs shipping PCs) who complain that this is fouling up their manufacturing line (or something of that ilk). Otherwise, our management typically says that the risks don't outweigh the benefits.

It's also costly in terms of time, effort, and testing for us to validate a modification to a released OS. We have a mindbogglingly massive amount of automated machinery dedicated to processing and validating the things that we check in while developing the current OS builds. But it's a special costly ask to spin up some to all of those activities to validate backported fixes. We do it all the time for Patch Tuesday, but in those patches, they only pass through the minimum number of fixes required to maximize the restoration of productivity/security/revenue/etc. because every additional fix adds additional complexity and additional risk.

So from our little team working hard to make developers happy, we virtually never make the cut for servicing. We're sorry, but we hope you can understand. It's just the reality of the situation to say "nope" when people ask for a backport. In our team's ideal world, you would all be running the latest console bits everywhere everytime we make a change. But that's just not how it is today.

Original Source: https://github.com/microsoft/terminal/issues/279#issuecomment-439179675

Why can't we have mixed elevated and non-elevated tabs in the Terminal?

guest speaker @DHowett-MSFT

[1] It is trivial when you are *hosting traditional windows* with traditional window handles. That works very well in the conemu case, or in the tabbed shell case, where you can take over a window in an elevated session and re-parent it under a window in a non-elevated session.

When you do that, there's a few security features that I'll touch on in [2]. Because of those, you can parent it but you can't really force it to do anything.

There's a problem, though. The Terminal isn't architected as a collection of re-parentable windows. For example, it's not running a console host and moving its window into a tab. It was designed to support a "connection" – something that can read and write text. It's a lower-level primitive than a window. We realized the error of our ways and decided that the UNIX model was right the entire time, and pipes and text and streams are *where it's at.*

Given that we're using Xaml islands to host a modern UI and stitching a DirectX surface into it, we're far beyond the world of standard window handles anyway. Xaml islands are fully composed into a single HWND, much like Chrome and Firefox and the gamut of DirectX/OpenGL/SDL games. We don't **have** components that can be run in one process (elevated) and hosted in another (non-elevated) that aren't the aforementioned "connections".

Now, the obvious followup question is *"why can't you have one elevated connection in a tab next to a non-elevated connection?"* This is where @sba923 should pick up reading (:smile:). I'm probably going to cover some things that you (@robomac) know already.

[2] When you have two windows on the same desktop in the same window station, they can communicate with each other. I can use `SendKeys` easily through `WScript.Shell` to send keyboard input to any window that the shell can see.

Running a process elevated *severs* that connection. The shell can't see the elevated window. No other program at the same integrity level as the shell can see the elevated window. Even if it has its window handle, it can't really interact with it. This is also why you can't drag/drop from explorer into notepad if notepad is running elevated. Only another elevated process can interact with another elevated window.

That "security" feature (call it what you like, it was probably intended to be a security feature at one point) only exists for a few session-global object types. Windows are one of them. Pipes aren't really one of them.

Because of that, it's trivial to break that security. Take the terminal as an example of that. If we start an elevated connection and host it in a *non-elevated* window, we've suddenly created a conduit through that security boundary. The elevated thing on the other end isn't a window, it's just a text-mode application. It immediately does the bidding of the non-elevated host.

Anybody that can *control* the non-elevated host (like `WScript.Shell::SendKeys`) *also* gets an instant conduit through the elevation boundary. Suddenly, any medium integrity application on your system can control a high-integrity process. This could be your browser, or the bitcoin miner that got installed with the `left-pad` package from NPM, or really any number of things.

It's a small risk, but it *is* a risk.

Other platforms have accepted that risk in preference for user convenience. They aren't wrong to do so, but I think Microsoft gets less of a "pass" on things like "accepting risk for user convenience". Windows 9x was an unmitigated security disaster, and limited user accounts and elevation prompts and kernel-level security for window management were the answer to those things. They're not locks to be loosened lightly.

Original Source: https://github.com/microsoft/terminal/issues/632#issuecomment-519375707

What's the difference between a shell and a terminal?

guest speaker @zadjii-msft

I think there might be a bit of a misunderstanding here - there are two different kinds of applications we're talking about here: * shell applications, like `cmd.exe`, `powershell`, `zsh`, etc. These are text-only applications that emit streams of characters. They don't care at all about how they're eventually rendered to the user. These are also sometimes referred to as "commandline client" applications. * terminal applications, like the Windows Terminal, gnome-terminal, xterm, iterm2, hyper. These are graphical applications that can be used to render the output of commandline clients.

On Windows, if you just run `cmd.exe` directly, the OS will create an instance of `conhost.exe` as the *terminal* for `cmd.exe`. The same thing happens for `powershell.exe`, the system will create a new conhost window for any client that's not already connected to a terminal of some sort. This has lead to an enormous amount of confusion for people thinking that a conhost window is actually a "`cmd` window". `cmd` can't have a window, it's just a commandline application. Its window is always some other terminal.

Any terminal can run any commandline client application. So you can use the Windows Terminal to run whatever shell you want. I use mine for both `cmd` and `powershell`, and also WSL:

It's not the Terminal's responsibility to remember the commands executed by a commandline client. That's the responsibility of the *shell*. How would the terminal remember commands executed by something like `emacs` or `vim`? Those are both applications where the user is typing input and hitting enter, like they would at a cmd prompt, but without something that resembles a command history.

Original Source: https://github.com/microsoft/terminal/issues/6500#issuecomment-670035468

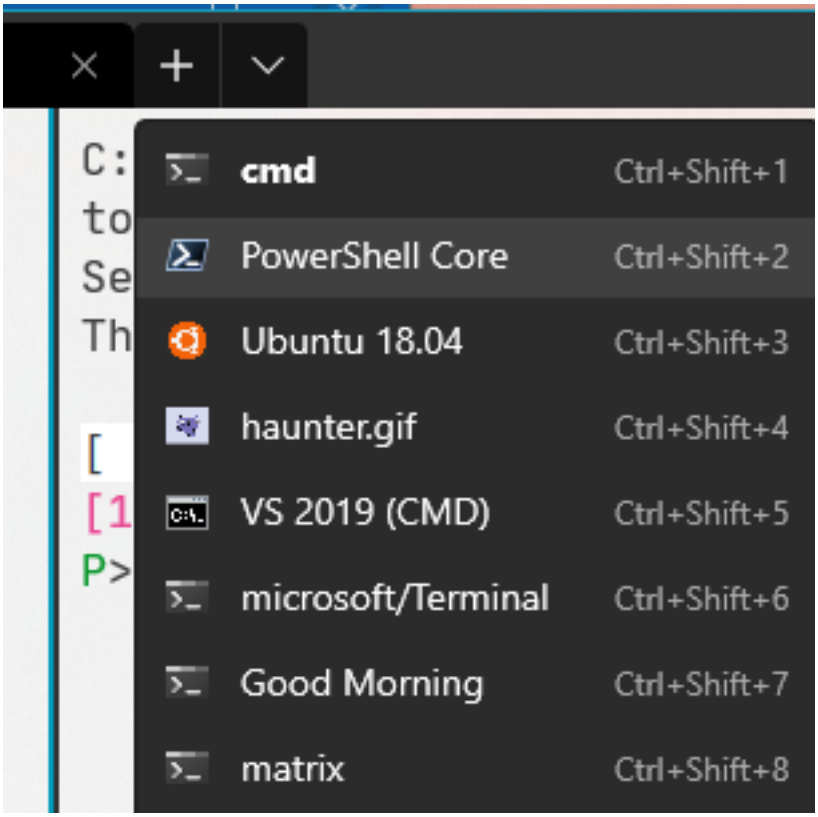

Figure 1: image